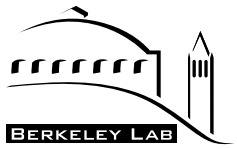


ERNEST ORLANDO LAWRENCE
BERKELEY NATIONAL LABORATORY

GenOpt(R)
Generic Optimization Program

Version 1.1.2

Simulation Research Group
Building Technologies Department
Environmental Energy Technologies Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

http://simulationresearch.lbl.gov

Michael Wetter
MWetter@lbl.gov

April 22, 2002

Notice:

This work was supported by the U.S. Department of Energy (DOE), by the Swiss Academy of Engineering Sciences (SATW), and by the Swiss National Energy Fund (NEFF).

Contents

1 Conventions

I $\mathbb{R}^n$ denotes the Euclidean space of n-tuplets of real numbers. Vectors $x \in \mathbb{R}^n$ are always treated as column vectors, and their elements are denoted by superscripts. Since no ambiguity can arise, we will write $x = (x^1, x^2, \ldots, x^n)$ to denote the column vector x, without a transpose sign. The inner product in $\mathbb{R}^n$ is denoted by $\langle \cdot, \cdot \rangle$ and defined by $\langle x, y \rangle \triangleq \sum_{i=1}^{n} x^i \, y^i$. The norm in $\mathbb{R}^n$ is denoted by $\| \cdot \|$ and is defined by $\|x\| \triangleq \langle x, x \rangle^{1/2}$.

II $f(\cdot)$ denotes a function where $(\cdot)$ stands for the undesignated variables. $f(x)$ denotes the value of $f(\cdot)$ at the point x. $f\colon A \to B$ indicates that the domain of $f(\cdot)$ is in the space A and its range in the space B.

2 Summary

The use of system simulation for analyzing complex engineering problems is increasing. Usually, a lot of time is spent on specifying the problem for a computer simulation. Once this has been done, the analyst usually does not attempt to optimize the design. One reason for this is that there is usually no time to go through the lengthy process of varying the input data, running the simulation and comparing the various results. Another reason that systems are not optimized is that they are often too complex, so that determining the optimal parameters is just not feasible without using a sophisticated algorithm.

To overcome these difficulties, GenOpt, a generic optimization program, has been developed. GenOpt does automatic optimization of a user-supplied objective function using search techniques that require minimum effort and time. Since one of the main application fields of the software is building system simulation, it has to consider the special characteristics of the simulation problems in this area, i.e.:

1. The number of free parameters is usually small (on the order of 10).
2. Much more computer time is spent evaluating the objective function than determining the new parameter set.
3. Analytical properties of the objective function are usually unavailable.

In order to make the program widely applicable, the following requirements need to be satisfied:

1. The program must be capable of minimizing a so-called black-box function (a function for which no analytical properties are available).
2. It must be possible to couple the optimization program to any simulation program that calculates the objective function on any operating system without having to modify or recompile either program.
3. The user must be able to choose an appropriate optimization algorithm from a library or easily implement a custom algorithm without having to recompile and understand the whole optimization environment.

GenOpt fulfills these requirements. To ensure platform independence, GenOpt is written entirely in Java. The coupling to a simulation program can be done by simply specifying in a configuration file how to exchange data and how to call the simulation. The implementation of the user's own optimization algorithms can be simply done by inheriting a superclass that offers methods to use the functionality of GenOpt. Hence, it is not necessary to know the overall program structure of GenOpt to implement and test a custom optimization algorithm.

Several test cases have shown that it is easy to couple a new simulation program, specify the optimization parameters and minimize the objective function. Therefore, in designing complex systems, as well as in system analysis for research purposes, a generic optimization program like GenOpt offers valuable services. However, you have to bear in mind that system optimization is not a trivial matter: The efficiency and success of an optimization is strongly affected by the properties and the formulation of the objective function, and by the proper selection of an optimization algorithm.

3 Introduction

Computer simulation is increasingly being used to analyze complex engineering problems. Computer analysis is generally required to account for the interactions and time-dependent effects of the system. Once the problem has been formulated (which can take a long time), the question arises whether the chosen values of the design parameters are such that the system is operating "optimally".

To optimize a system, the optimal values of a set of input parameters must be found. However, these values are usually not obvious. For example, consider the design of a low-energy office building where we want to reduce the energy for cooling the building. Suppose we have a given energy flux (i.e., solar radiation, internal heat from lighting, computers, etc.) that leads to an increase in room temperature during the day such that cooling has to be provided. Suppose we want to decrease the cooling energy by pre-cooling the building at night by circulating chilled water in concrete ceilings. The water is cooled by a cooling tower that uses low nighttime air temperature ("free-cooling" system). However, how to operate this system is not obvious. You could run it for a short time with a high water flow (when the outside air temperature is lowest and so cooling the circulated water is most efficient), or you might run it over a longer period with a lower water flow (to save pump energy) but also a lower cooling tower efficiency due to the warmer mean outside air temperature. The optimal operating point probably lies somewhere between those two extremes.

Alternatively, you might want to minimize the yearly cost of the cooling system. Assume that the cost is known as a function of the installation cost and the operation cost. Suppose you know the installation cost of the cooling tower as a function of its size and the yearly energy cost as a function of the energy consumption (which is evaluated in the simulation). Then you can use GenOpt to design the system for lowest yearly operating cost.

Of course, optimization is not limited to problems in building energy systems. The general structure of a problem is always the same: You specify the values of a set of parameters for a simulation and you get a result back. The optimization itself does not care what those parameters stand for, nor does it care what the result of the simulation represents. Therefore, it is possible to apply optimization methods to a wide range of physical and engineering problems, including regression analysis. The remaining chapters in this report are as follows: Chapter 4 briefly explains what GenOpt does. Chapter 5 describes the general structure of an optimization problem and shows how optimization problems are treated in GenOpt. Chapter 6 describes some important algorithms for unconstrained non-linear optimization and Chapter 7 shows how constraints can be taken into account. In Chapter 8 it is shown how new optimization algorithms can be implemented in GenOpt. Chapter 9 shows how to install and run GenOpt. Finally, Chapter 10 describes how to configure GenOpt to optimize a given objective function.

4 What is GenOpt?

GenOpt is a generic optimization program for multidimensional minimization of an objective function that is computed by an external program. GenOpt automatically finds the values of selected free parameters (the independent variables) that minimize the objective function.

GenOpt can be coupled to any simulation program that reads its input from one or more text files and writes its output (i.e., the value of the objective function) to a text file. GenOpt is written entirely in Java so that it is completely platform independent. The platform independence and the general interface make GenOpt applicable to a wide range of optimization problems.

To do the optimization, GenOpt offers a library with multi-dimensional and one-dimensional optimization algorithms, as well as an algorithm for doing parametric runs on an orthogonal, equidistant grid. GenOpt's structure is open, so that new optimization algorithms can easily be added without knowing the details of the program structure. A base class serves as an interface between the kernel of GenOpt (which is used for reading and writing the files, storing the results, etc.) and the actual optimization algorithms. This class delivers all methods required to access the data that specifies the optimization process, reporting results to the result database, executing the simulation program, etc.

5 Optimization Problem

5.1 General Definition of an Optimization Problem

An optimization problem consists of

1. A set of free parameters (the independent variables, also called design parameters).
2. Some constraints that bound the domain of the free parameters and dependent variables.
3. An objective function (the function to be minimized) that depends on the free parameters.

Without loss of generality, optimization can be considered as minimizing a function since maximization can always be translated into minimization by simply changing the sign of the objective function. Therefore, we consider here only the case of minimization. Hence, the general optimization problem can be formulated as

$$\min_{x \in \mathbf{X}} f(x) \tag{5.1a}$$

$$\mathbf{X} \triangleq \{x \in \mathbb{R}^n \mid h(x) = 0,\, g(x) \leq 0\} \tag{5.1b}$$

where $f\colon \mathbb{R}^n \to \mathbb{R}$, $h\colon \mathbb{R}^n \to \mathbb{R}^m$, and $g\colon \mathbb{R}^n \to \mathbb{R}^p$. Each component of $h(\cdot)$ and $g(\cdot)$, respectively, represents one constraint. $\mathbb{R}^n$ denotes the n-dimensional

space of real numbers. In case of unconstrained optimization, (5.1) reduces to

$$\min_{x \in \mathbb{R}^n} f(x), \tag{5.2}$$

with $f: \mathbb{R}^n \to \mathbb{R}$.

5.2 Problem Specification for GenOpt

Since for optimization in building simulation most of the constraints on free parameters can be formulated as box constraints (i.e., where the lower and upper bound of the free parameter is a constant), a default scheme for such constraints is implemented in GenOpt. If more complex constraints have to be specified, it is possible to include those constraints into the objective function by adding penalty functions, barrier functions or slack variables (see Section 7.2, page 42) and hence covering constraints of the problem type specified by (5.1).

Therefore, the problem specification for a GenOpt optimization is

$$\min_{x \in \mathbf{X}} f(x) \tag{5.3a}$$

$$\mathbf{X} \triangleq \{x \in \mathbb{R}^n \mid l^i \leq x^i \leq u^i;\ i \in \{1, 2, \ldots, n\};\ l, u \in \mathbb{R}^n\} \tag{5.3b}$$

where $f: \mathbb{R}^n \to \mathbb{R}$ is evaluated by any external computer program.

5.3 Properties of Optimization Problems

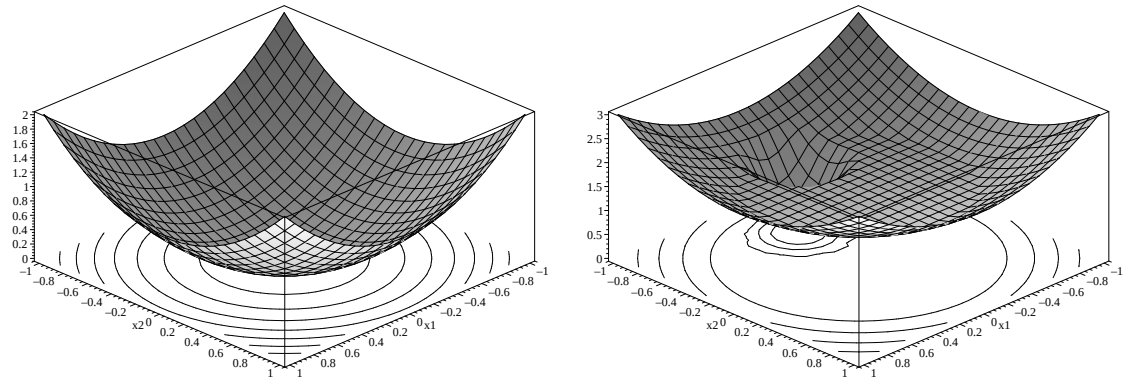
Most optimization problems can be formulated as nonlinear constrained problems. However, it is advisable – and in some cases even necessary – to take advantage of some properties of the problem. It is obvious that (a) no optimization algorithm works best on all possible functions $f: \mathbb{R}^n \to \mathbb{R}$ and (b) no optimization algorithm can guarantee to find the global minimum if local minima exist.

The selection of the optimization algorithm depends primarily on the following considerations:

- structure of the function (linear, non-linear, convex, continuous, number of local minima, etc.)
- availability of analytic first and second derivatives
- size of the problem (number of independent parameters)
- problem constraints (on the independent parameters and/or the dependent variables)

The need to select an algorithm that works efficiently on a particular problem leads to a large number of available optimization methods. Therefore, it is valuable to have a general framework that allows easy implementation of optimization algorithms into an environment that launches a simulation and gets back the required function value.

One has to bear in mind that this approach makes it impossible to get analytical information about the objective function. However, most of the applications in building simulation, which is a main target of GenOpt, cannot be expressed analytically.



(a) Unimodal function

(b) Function with null-space

Fig. 5.1: Example functions

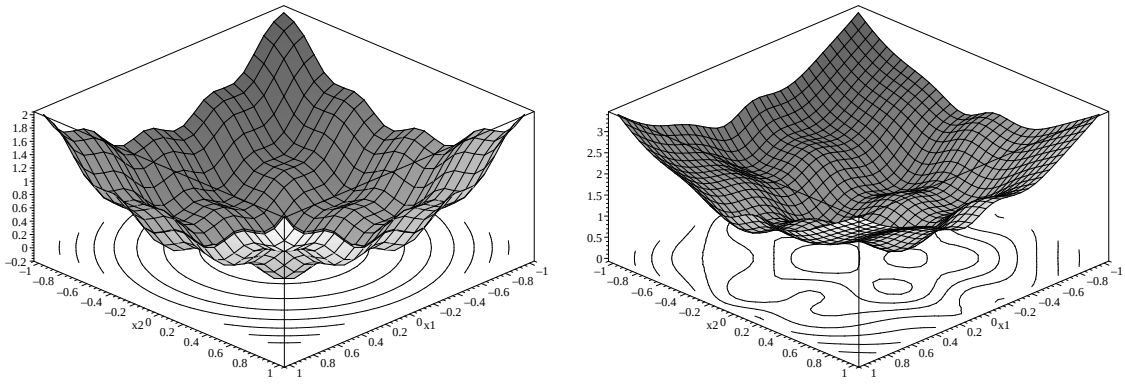
5.4 Structure of the Objective Function

Fig. 1 to Fig. 6 show some possible structures of a 2-dimensional function in order to demonstrate that it is impossible to have a "perfect" optimization algorithm that works well on all problems. The different structures allow – or in some cases even require – special techniques to find the minimum and to increase the efficiency of the optimization:

Unimodal functions (Fig. 5.1(a)) For unimodal functions it holds that if and only if $x^* \in \mathbb{R}$ is a minimum of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, then along each direction $d \in \mathbb{R}^n$, starting from x^*, the function value increases strictly. Unimodal functions, therefore, have one and only one minimum. Thus, if one reaches a minimum, it is for sure the global minimum. If the function can be well approximated by a quadratic function, special techniques can be applied that ensure a very efficient minimization, particularly if the first derivatives (gradient) and second derivatives (Hessian matrix) of the function are known. In addition to the structure of the function, the number of free parameters plays a crucial rule in selecting an efficient optimization algorithm.

Functions with a significant null-space effect (Fig. 5.1(b)) For this class of functions, perturbations of free parameters have no effect or only a small effect on the function value if one is in a flat or almost flat area of the function. No improvement or only little improvement is achieved and hence the optimization might stop (due to the stopping criteria) without reaching the minimum. One can avoid such problems by (I) eliminating correlated parameters, (II) eliminating parameters on which the function show only small sensitivity, (III) reformulating the function, (IV) selecting another algorithm, or (V) selecting another starting point for the optimization. Point (IV) and (V) are a very insecure method if the properties of the function and the optimization algorithm are not known.

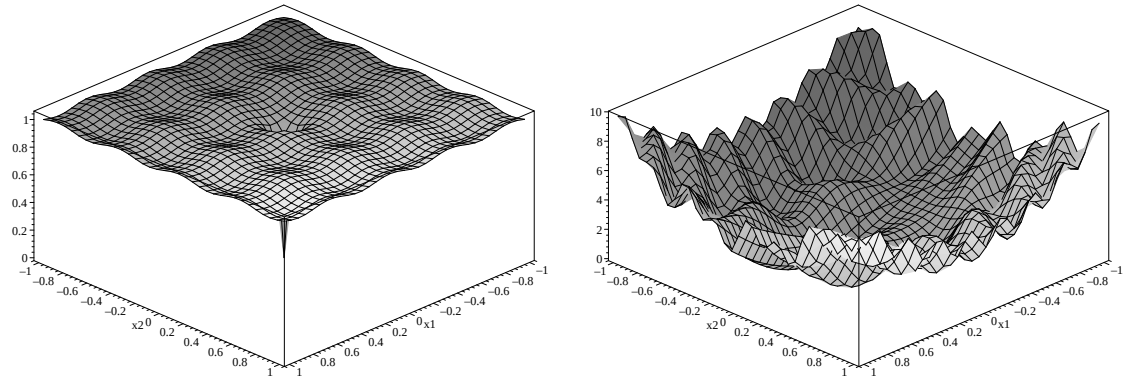
Functions with a non-nice surface (Fig. 5.2(a)) In this case, gradient-based search methods act very poorly. The local information that is gathered in the neighborhood of the current point for evaluating the



(a) Function with a non-nice surface

(b) Function with a few local minima

Fig. 5.2: Example functions



(a) Function without useful global information

(b) Function with a large number of local minima

Fig. 5.3: Example functions

gradient indicates nothing about the shape of the function outside this neighborhood. If even parasitic local minima exist, the objective function must be smoothed to avoid being trapped in local minima. These local minima are often the result of some noise in the objective function, possibly caused by numerical solvers.

Functions with a few local minima (Fig. 5.2(b)) In this case, one might get trapped in one of the local minima. Selecting other starting points for the iteration might help find the global minimum.

Functions without useful global information (Fig. 5.3(a)) If the function does not have a useful structure that helps finding the global minimum, a very costly exhaustive search over the whole domain where the minimum is expected to lie has to be done.

Functions with a large number of local minima (Fig. 5.3(b)) Selecting different starting points with usual optimization algorithms is not suc-

cessful for such cases. A search for the global minimum has to be made with methods like Simulated Annealing, Tabu Search or Genetic Algorithm.

In general, it can be said that if a point $x^* \in \mathbb{R}^n$ is a minimizer of a function $f : \mathbb{R}^n \to \mathbb{R}$, and $f(\cdot)$ is quasi-convex, i.e., if for all $x_1, x_2 \in \mathbb{R}^n$ and for all $\lambda \in [0, 1]$

$$f(\lambda x_1 + (1 - \lambda) x_2) \leq \max\{f(x_1), f(x_2)\}, \tag{5.4}$$

then x^* is a *global* minimizer.

If the objective function has several minima, then there is no guarantee that one has reached the global minimum unless additional properties of the function are known. Algorithms like Simulated Annealing or Genetic Algorithms increases the chance of getting to the global minimum but they do not guarantee finding the global minimum. Furthermore, the higher probability of reaching the global (or at least a better local) minimum is paid for with higher computation time. However, bear in mind that:

1. without automatic optimization it is time consuming and expensive (and often impossible) to find even a local minimum, and
2. a local minimum is still a better solution to the problem than doing no optimization at all.

In summary, the selection of an optimization algorithm depends on the structure of the objective function and the number of free parameters. Unless certain properties of the function are known one can never be sure to have reached the global minimum. But even without the certainty of having found the optimal solution, the solution has at least been improved.

6 Optimization Algorithms

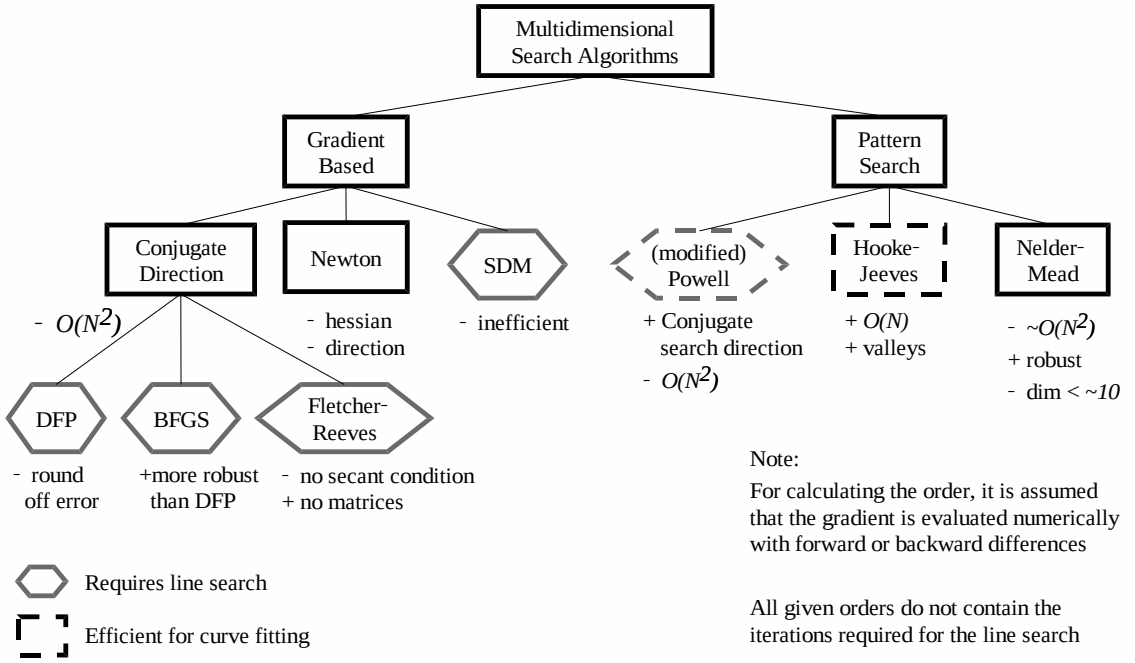


Fig. 6.1: Overview of some algorithms for non-linear unconstrained optimization

The fact that no analytical properties of the objective function, $f : \mathbb{R}^n \to \mathbb{R}$, are available limits the techniques that can be applied efficiently to seek a minimum of $f(\cdot)$. The solution of the problem has to be obtained completely numerically and, furthermore, it is difficult and time-consuming to obtain information about the properties of the objective function.

Fig. 6.1 gives an overview on some algorithms that solve non-linear, unconstrained problems of the form (5.2). The algorithms are divided into *gradient based* methods and *pattern search* methods. A brief overview about the main ideas and properties of the algorithms will be given below, where we assume that $f(\cdot)$ is sufficiently smooth.

Readers who are already familiar with the algorithms might skip this section. The algorithms that are in GenOpt's library are described on page 19.

6.1 Gradient Based Methods

Gradient based methods use the gradient (at the current iteration point) of the objective function, $f\colon \mathbb{R}^n \to \mathbb{R}$, with respect to the free parameters, that is

$$\nabla f(x) = \left(\frac{\partial f(x)}{\partial x^1}, \frac{\partial f(x)}{\partial x^2}, \ldots, \frac{\partial f(x)}{\partial x^n} \right)^T \tag{6.1}$$

to gather information about the function structure and to determine the direction of the next iteration step. Those methods can easily fail if the objective function is not differentiable or if it even has discontinuities.

The Newton method (also called Newton-Raphson method) and the steepest descent method (SDM) are two gradient based methods. Both are not used alone for optimization. However, they act as a base of a large class of optimization algorithms. Therefore, it is important to understand their main concepts.

6.1.1 Newton's Method

The Newton method is based on the 1-st order necessary condition for optimality, that is, if $x^* \in \mathbb{R}^n$ is a local minimum point, then

$$\nabla f(x^*) = 0. \tag{6.2}$$

The linear approximation to the gradient at x^* around a point $x \in \mathbb{R}^n$ is

$$\nabla f(x^*) = \nabla f(x) + H(x)\, d \tag{6.3}$$

where $H(x) \in \mathbb{R}^{n \times n}$ is the *Hessian* matrix

$$H(x) \triangleq \nabla^2 f(x), \tag{6.4}$$

with

$$H^{i,j}(x) \triangleq \frac{\partial^2 f(x)}{\partial x^i \, \partial x^j}, \tag{6.5}$$

and the vector $d \in \mathbb{R}^n$ is defined as

$$d \triangleq x^* - x. \tag{6.6}$$

Hence, the new step that has to be taken is

$$d = -H(x)^{-1} \nabla f(x). \tag{6.7}$$

Since (6.2) also holds for a maximum point, the Newton method does not know whether it is heading to a maximum or a minimum of $f(\cdot)$. Furthermore, $f(\cdot)$ must be twice continuously differentiable and the Hessian nonsingular. It should be mentioned that the numerical approximation of the Hessian is very costly and error-prone, particularly if the scale of the problem is not known.

Since Newton's method is a quadratic approximation of the objective function, it reaches the extremum of $f(\cdot)$ in one step if $f(\cdot)$ is a quadratic function.

Newton's method converges very fast if one is close to the optimum since the error of the quadratic function approximation is usually small close to the optimum. However, convergence to a local minimum point cannot be proven since the Newton method may lead to a maximum point.

6.1.2 Steepest Descent Method

If one uses only the first term of the Taylor series

$$f(x_{i+1}) - f(x_i) = \langle \nabla f(x_i),\, d_i \rangle + \frac{1}{2} \langle d_i,\, H(x_i)\, d_i \rangle + \dots \tag{6.8}$$

then the biggest reduction of $f(\cdot)\colon \mathbb{R}^n \to \mathbb{R}$ is obtained by choosing a direction $p_i \in \mathbb{R}^n$ that is a positive scalar of $d_i \in \mathbb{R}^n$, so that the inner product $\langle \nabla f(x_i), p_i \rangle$ has the highest possible negative value. This leads to the *steepest descent method* (SDM), where the direction $p_i$is calculated as

$$p_i = -\nabla f(x_i) \tag{6.9}$$

and the step d_i as

$$d_i = \alpha_i\, p_i, \qquad \alpha_i > 0. \tag{6.10}$$

The new value of the objective function is then

$$f(x_{i+1}) = f(x_i + d_i). \tag{6.11}$$

In the SDM algorithm, the step length is chosen so that α_i minimizes $f(x_i + \alpha_i\, p_i)$ along the direction p_i. Therefore, one has now only to solve the one-dimensional problem

$$\min_{\alpha_i > 0} f(x_i + \alpha_i\, p_i), \tag{6.12}$$

and not the original n-dimensional problem.

For solving this one-dimensional problem, a class of algorithms exists. They are generally referred to as *line-search algorithm*. If one has reached the minimum of (6.12) by performing a line search, one has not yet reached the minimum of $f(x)$. The whole process is repeated iteratively until a stopping criterion is satisfied.

It can be shown that the SDM algorithm generates orthogonal search directions. At the beginning, $f(\cdot)$ is reduced relatively fast, but close to the minimum the convergence rate gets very poor. The SDM algorithm has slow convergence since it uses no history of the previous iterations and only a linear approximation of the objective function.

6.1.3 Conjugate Direction Methods

A conjugate direction method is an approach that generates search directions that are conjugate to previous directions with respect to the Hessian matrix of the objective function.

Definition 6.1.1 (Conjugate Direction) *A set of vectors $\{p_i\}_{i=1}^n$ is mutually conjugate with respect to a matrix $Q \in \mathbb{R}^{n \times n}$, if and only if for all $i \neq j$*

$$\langle p_i,\, Q\, p_j \rangle = 0. \tag{6.13}$$

It can be shown that conjugate direction methods that minimize $f\colon \mathbb{R}^n \to \mathbb{R}$ along each direction p_i, $i \in \{1, 2, \ldots, n\}$ reach the minimum of quadratic functions of the form

$$f(x) = a + \langle b, x\rangle + \frac{1}{2}\langle x, Q\,x\rangle \tag{6.14}$$

where $Q \in \mathbb{R}^{n\times n}$ is a positive definite and symmetric matrix in maximal n steps (for a proof see [Wal75]). The following property makes conjugate direction methods widely used: Suppose that no additional evaluations of $f(\cdot)$ are required to determine the Hessian of $f(\cdot)$, that is, Q is either known analytically or can be determined with any method. Suppose further that the gradient of $f(\cdot)$ is determined *exactly* by forward or backward difference with n additional function evaluations. Then, not counting the function evaluations for the line search, the algorithm reaches the global minimum of $f(\cdot)$ defined by (6.14) with a maximum of n^2 function evaluations if the line search (6.12) is exact.

This property is very powerful since it gives fast convergence if one is already close to the minimum, that is, if (6.14) is a good approximation for the function being optimized. Since the conjugate direction methods are highly sensitive to numerical errors, it is recommended to restart the conjugate direction methods with an SDM step after n iterations [Kar89].

6.1.4 Davidon-Fletcher-Powell

The Davidon-Fletcher-Powell method (DFP) is a hybrid method that is based on the Newton method and the SDM.

For constructing the DFP method, it is assumed that a matrix $D_i \in \mathbb{R}^{n\times n}$ satisfies the *inverse secant condition*, that is

$$D_i\left(\nabla f(x_{i+1}) - \nabla f(x_i)\right) = x_{i+1} - x_i, \tag{6.15}$$

where D_i is a positive definite and symmetric matrix that approximates the Hessian. The search direction $p_i \in \mathbb{R}^n$ is obtained from (6.15) assuming that $\nabla f(x_{i+1}) = 0$. That leads to a new search direction

$$p_i = -D_i\,\nabla f(x_i). \tag{6.16}$$

Along the search direction p_i, a line-search that minimizes

$$h(\alpha_i) \triangleq f(x_i + \alpha_i\,p_i) \tag{6.17}$$

is carried out in order to obtain the new point x_{i+1}.

Updating the matrix D_i to D_{i+1} is done such that

1. No matrix inversion is necessary.
2. No additional function evaluations are required.
3. The generated matrix D remains symmetric and positive definite if one starts with a symmetric positive definite matrix.
4. The generated search directions are mutually conjugate.

In summary, the computationally costly operations of the DFP method are the evaluation of the gradient and the line search for determining α_i. However, the DFP method has a tendency to produce D_i's that are not positive definite, primarily due to inaccuracy in the line search, but also due to round-off errors [Sch94]. To overcome this problem, the DFP method is restarted after $(n+1)$ steps with an SDM step. The tendency to generate D_i's that are not positive definite leads to the replacement of the DFP method by the *Broyden-Fletcher-Goldfarb-Shanno* method (BFGS), which does not have this drawback. For a further discussion of the DFP method, see [PSU88].

6.1.5 Broyden-Fletcher-Goldfarb-Shanno

The *Broyden-Fletcher-Goldfarb-Shanno* method (BFGS) shares the same properties as mentioned above for the DFP method, except that the *secant condition*

$$D_i\,(x_{i+1} - x_i) = \nabla f(x_{i+1}) - \nabla f(x_i) \tag{6.18}$$

is fulfilled instead of the inverse secant condition. This leads to the requirement that a linear system has to be solved in order to find the direction $p_i \in \mathbb{R}^n$, i.e.,

$$D_i\, p_i = -\nabla f(x_i). \tag{6.19}$$

Since the matrix D_i is positive definite due to its initial form and its update scheme, (6.19) always has a unique solution.

The BFGS method does not share the DFP tendency of generating D_i's that are not positive definite. In fact, Dennis and Schnabel state that it is the "best" currently known update for the Hessian matrix [DS83]. For further details of the BFGS method, see [PSU88].

6.1.6 Fletcher-Reeves

The Fletcher-Reeves algorithm is based on a recurrence formula that generates a sequence of conjugate directions without using matrix calculations. To start the algorithm, an SDM step is made. In subsequent steps, each direction is determined by

$$p_i = -\nabla f(x_i) + \frac{\langle \nabla f(x_i),\, \nabla f(x_i) \rangle}{\langle \nabla f(x_{i-1}),\, \nabla f(x_{i-1}) \rangle}\, p_{i-1} \tag{6.20}$$

whereas a line search according to (6.12) is carried out to move from x_i to x_{i+1}.

Due to the conjugate direction, this method also terminates after at most n steps for quadratic functions of the form (6.14) assuming theoretically that the computations are carried out without any error.

Since no linearly-independent search directions are left over after n iterations, the algorithm is restarted after each n iterations with an SDM step discarding all previous experience that would be transmitted in the calculation of p. However, provided that the restart is not done more frequently than every n iterations, the property of quadratic convergence remains. It turns out that after the n iterations, it is beneficial to do one more step before restarting with an SDM step. The last iteration then compensates for the accumulation of the

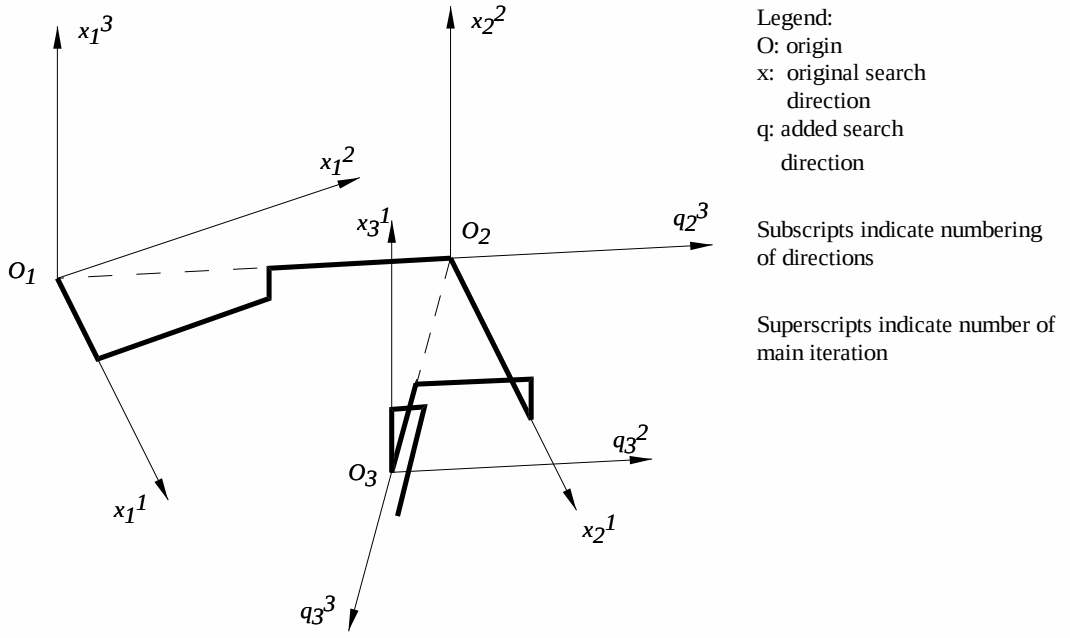


Fig. 6.2: Trajectory of Powell's method

errors of the first n iterations [FR64].

In contrast to the DFP and BFGS methods, the Fletcher-Reeves algorithm does not use the secant condition. The main advantage of the Fletcher-Reeves algorithm over DFP and BFGS is that it does not use any matrix calculations, which is of interest for large-scale problems. However, since the number of independent parameters is usually small in building system optimization, a matrix formulation is not a drawback. For further discussion and for a proof that (6.20) generates conjugate directions, see [FR64] and [Wal75].

6.2 Pattern Search

For a description of the Simplex and the Hooke-Jeeves algorithm, see Chapter 6.3. [LTT00] discusses different pattern search algorithms. For a proof of convergence of pattern search algorithms, see [Tor97] and [LT99].

6.2.1 Modified Powell

The conjugate direction methods that are presented above all require knowledge of the gradient of $f: \mathbb{R}^n \to \mathbb{R}$. However, in building simulation, the gradient does usually not exist since the simulation model is in most cases not even continuous with respect to the design parameter, x.

Assuming that the gradient exists but is not available analytically, it has to be approximated numerically. However, the numerical approximation is not trivial. It cannot simply be done by a forward or backward approximation of the partial derivatives – which costs n function evaluations in each iteration

step – particularly if the scaling of the function is not known. Furthermore, the conjugate direction methods presented above are sensitive to error in the gradient approximation.

Powell has overcome these difficulties by formulating an algorithm that does not use the gradient but still generates conjugate search directions. It starts with a set of n orthogonal unit vectors, say $\{x_1^i\}_{i=1}^n$ according to Fig. 6.2 (where $n = 3$). Along each of the n directions, a line search is carried out that minimizes $f(\cdot)$.

After the last line search, a line search along the resulting direction q_2^3 is done. The direction that leads to the biggest improvement in minimizing $f(\cdot)$ (in our case x_1^2) is regarded as "exhausted" and – if particular conditions are fulfilled – is replaced by the resulting search direction q_2^3. The whole process is now restarted using the vectors x_2^1, x_2^2 and q_2^3 as the base direction for the line search.

After n main iterations – each consisting of $(n + 1)$ sub iterations – a set of mutually conjugate directions is obtained and one is at the global minimum provided that $f(\cdot)$ is of the form of (6.14), the line search is done exactly, and q replaces all base search directions x^i, $i \in \{1, 2, \ldots, n\}$. However, whether the set of generated directions is conjugate depends highly on the quality of the line search.

The original scheme that strictly exchanges the "exhausted" direction after every set of sub iterations might lead to some directions being adopted more often. Hence, the old set of directions is retained longer, which might cause slow convergence for problems with many variables. This is a serious problem if the number of free parameters is bigger than five. This requires a modification of the direction exchange scheme. With the modified exchange scheme, one of the mutually conjugate directions might be thrown away leading to more than n main iterations in order to find the minimum of (6.14). However, this modification turns to be essential for minimizing a function of twenty variables [Pow64].

Powell's method has fast convergence if $f(\cdot)$ is a sum of squares of functions, as is often the case in data fitting [Pow65]. If the number of free parameters is bigger than 10 or 20, the modified Powell algorithm should not be used [Sch94].

Though the convergence properties of Powell's method are good, it is generally accepted that quasi-Newton methods like BFGS (with approximation of the gradient) have faster convergence than non-derivative methods [GMW81, p. 131]. However, it has to be stressed that numerical differentiation is not an easy task. It requires a sophisticated method in order to reduce errors that can strongly affect the convergence properties of the quasi-Newton method.

For a further discussion of the algorithm, see [Pow64] and [Sch94].

6.3 Implemented Algorithms

6.3.1 Hooke-Jeeves

Hooke and Jeeves [HJ61] developed a pattern search method that generates steps along the valley of the objective function. The algorithm requires neither the gradient of the objective function nor a line search. The original algorithm solves the unconstrained problem (5.2). In GenOpt, the algorithm has been modified to solve the box-constraint problem, (5.3), directly, i.e., without doing a parameter transformation as the one of Section 7.1.1 on page 41.

The Hooke-Jeeves pattern search technique is based on the assumption that it is worthwhile to make further steps in a direction that proved to be successful in earlier steps. It starts with small orthogonal exploratory steps in each direction. After exploring each direction, it assumes that it is likely to get a further improvement in the direction that results from previous successful explorations. Thus, it skips the exploratory moves and makes a further step in this direction. At the new point, exploratory moves in each direction are carried out again. This ensures that the search stays in the valley of the objective function. If no further improvement can be achieved, the algorithm restarts from the last successful base with smaller exploratory steps. Otherwise, it takes another step in the resulting direction, followed by exploratory steps. Hooke and Jeeves found empirically that the number of function evaluations increases only linearly with the number of independent variables, i.e., is $\mathcal{O}(n)$ [HJ61], where n denotes the number of independent variables. This might be explained by the fact that a valley is actually a one-dimensional object and, hence, the dimensionality of the problem can be reduced if one searches along a valley. The method has been successfully used for curve fitting [HJ61].

Further discussion of this method can be found in [Wil64, Avr76, Wal75].

Audet and Dennis [AD00] show that if the objective function $f : \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable, and has bounded level sets, then the Hooke-Jeeves algorithm converges to a point $x^* \in \mathbb{R}^n$ that satisfies $\|\nabla f(x^*)\| = 0$.

a) Algorithm Modifications

Smith [Smi69] reports that applying a universal step size for each variable causes some parameters to be essentially ignored during much of the search process. Therefore, he proposed to initialize the step size for each variable by

$$\Delta x^i = \delta |x_0^i|, \tag{6.21}$$

where $\delta > 0$ is a fraction of the initial step length and $x_0^i \in \mathbb{R}$ the starting point of the i-th design parameter. In GenOpt's implementation, (6.21) is not used. The initialization of Δx^i is done by using the value of the parameter `Step` specified in the command file (see page 65), which also allows taking the scaling of the problem into account.

In the original implementation, the search of the exploration move is always done first in the positive, then in the negative direction along the basis vectors, $e_i \in \mathbb{R}^n$, $i = 1, \ldots, n$. Bell and Pike [BP66] proposed searching first in the direction that led in the last exploration move to a reduction of the

objective function. This increases the probability to reduce the objective function already by the first exploration move, thus allows skipping the second trial.

De Vogelaere [DV68] proposed changing the algorithm such that the maximum number of function evaluations cannot be exceeded, which can be the case in the original implementation.

All three modifications are implemented in GenOpt.

The Hooke-Jeeves algorithm solves the unconstrained problem $\min_{x\in\mathbb{R}^n} f(x)$. Therefore, we convert the box-constrained problem

$$\min_{x\in\mathbf{X}} \tilde{f}(x) \tag{6.22a}$$

$$\mathbf{X} \triangleq \{x \in \mathbb{R}^n \mid l^i \le x^i \le u^i;\ l^i, u^i \in \mathbb{R} \cup \{\pm\infty\};\ i \in \{1, 2, \ldots, n\}\} \tag{6.22b}$$

where $\tilde{f}\colon \mathbb{R}^n \to \mathbb{R}$, to the form

$$\min_{x\in\mathbb{R}^n} f(x) \tag{6.23a}$$

where

$$f(x) \triangleq \begin{cases} \tilde{f}(x), & \text{if } x \in \mathbf{X} \\ \infty, & \text{otherwise.} \end{cases} \tag{6.23b}$$

By using this transformation, we can ensure convergence to a stationary point if the objective function is continuously differentiable and has bounded level sets [AD00].

b) Algorithm Description

The algorithm can be divided into an *initial exploration* (I), a *basic iteration* (II), and a *step size reduction* (III). (I) and (II) make use of so-called *exploratory moves* in order to get local information about the direction in which the function decreases.

The exploratory moves are executed as follows (see Fig. 6.3):
Let $\Delta x^i \in \mathbb{R}$ be the step size of the i-th design parameter, and $e_i \in \mathbb{R}^n$ the unit vector along the x^i axis. Assume we are given a base point, called the resulting base point, x_r, and its function value, say $f_p = f(x_r)$. Then we perform a sequence of orthogonal exploratory moves. To do so, we start with the first direction (i.e., $i = 0$) and set the new point

$$x_r \leftarrow x_r + \Delta x^i\, e_i. \tag{6.24}$$

Provided that x_r is feasible, that is $l^i \le x_r^i \le u^i$ for all $i \in \{1, \ldots, n\}$, we evaluate the objective function $f_r \leftarrow f(x_r)$. If $f_r < f_p$, then the new point becomes the resulting base point, and we assign

$$f_p \leftarrow f_r. \tag{6.25}$$

Otherwise, we set

$$\Delta x^i \leftarrow -\Delta x^i, \tag{6.26}$$

$$x_r \leftarrow x_r + 2\,\Delta x^i\, e_i, \tag{6.27}$$

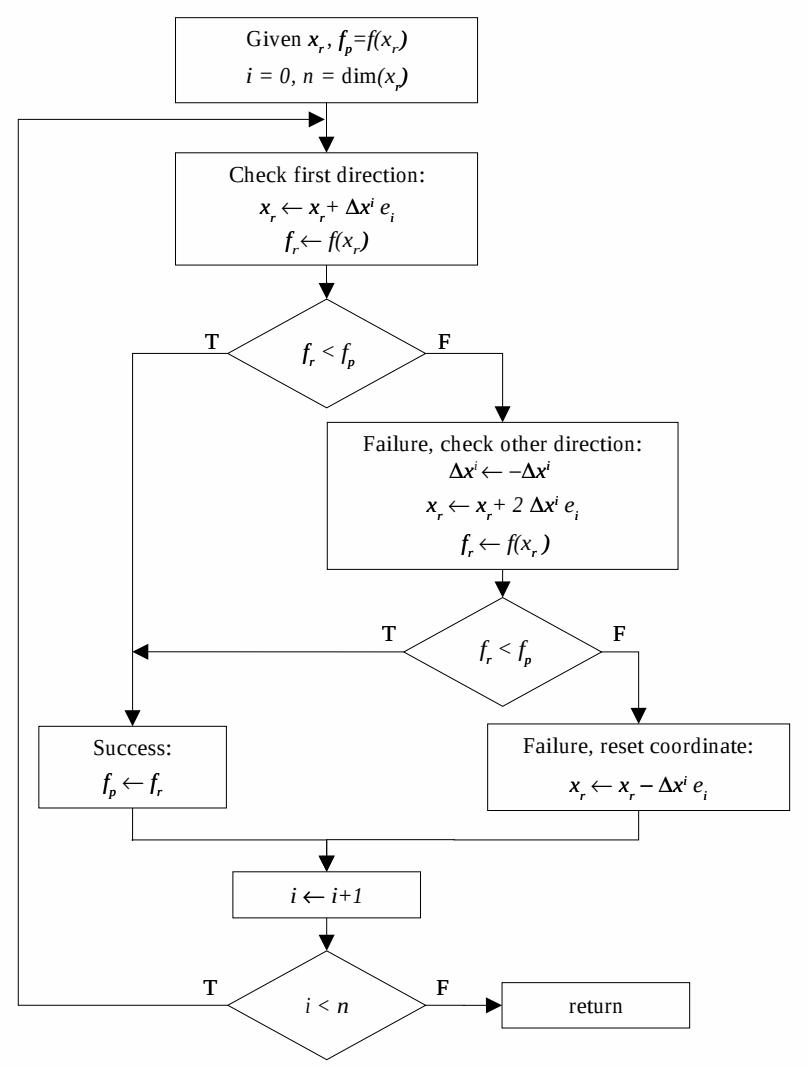


Fig. 6.3: Flow chart of exploration move, $E(x_r,\ f_p)$

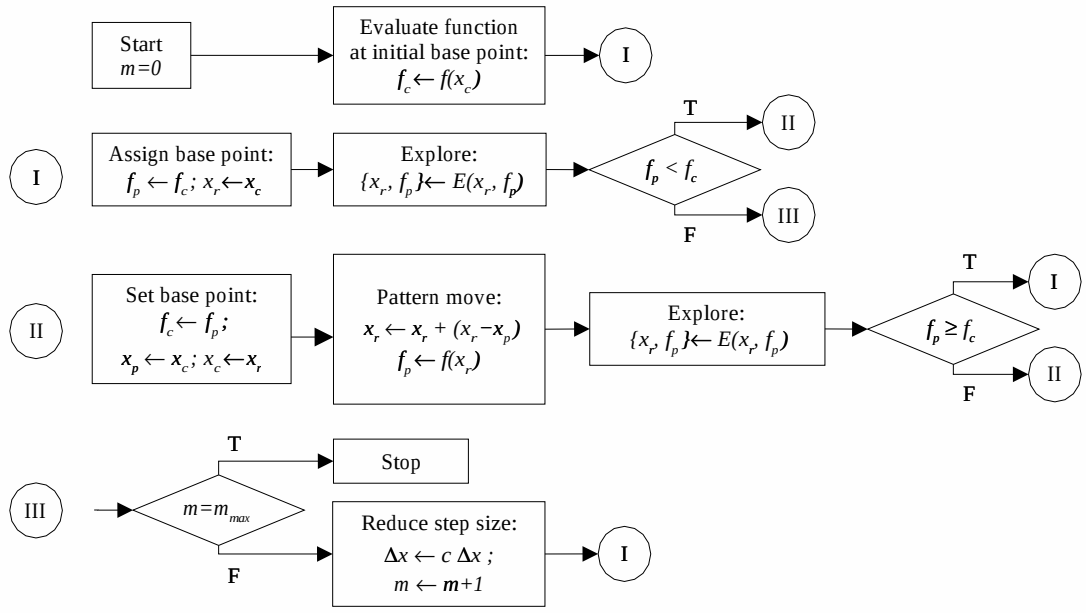


Fig. 6.4: Flow chart of Hooke-Jeeves algorithm

and evaluate and assign again $f_r \leftarrow f(x_r)$. If this exploration is successful, we apply (6.25). If it fails, we reset the resulting base point by

$$x_r \leftarrow x_r - \Delta x^i \, e_i \tag{6.28}$$

so that the resulting base point has not been altered by the exploration in the direction along e_i. Therefore, if any of the exploration moves have been successful, we have a new resulting base point, x_r, and a new function value $f_p = f(x_r)$. Using the (probably new) resulting base point, x_r, the same procedure is repeated along the next dimension, e_{i+1}, until an exploration along all base vectors, e_i, has been done. Note that, according to (6.27), Δx^i has in the next exploration move along e_i the sign that led in the last exploration to a success (if any direction was successful).

At the end of the n exploratory moves, we have a new resulting base point, x_r, if and only if at least one of the exploratory moves led to a reduction of the objective function.

(I) Initial Iteration
In the initial iteration, we have a current base point, x_c. We assign $x_r \leftarrow x_c$ and make the exploration moves around x_r. If at least one of the exploration move leads to a reduction of the objective function, then we go to the basic iteration (II), otherwise we reduce the step size according to (III).

(II) Basic Iteration
We update the function value of the base point by assigning

$$f_c \leftarrow f_p \tag{6.29}$$

and assign to the previous base point, x_p, the value of the current base point, x_c, and to the current base point, x_c, the value of the resulting base point, x_r, i.e.,

$$x_p \leftarrow x_c, \tag{6.30}$$
$$x_c \leftarrow x_r. \tag{6.31}$$

Then, we make a pattern move, given by

$$x_r \leftarrow x_r + (x_r - x_p) \tag{6.32}$$

Now, we assign $f_p \leftarrow f(x_r)$. Regardless of whether the pattern move leads to a reduction of the objective function, we do exploratory moves around x_r. If any of the exploratory moves is successful, then x_r and consequently $f_p = f(x_r)$ are altered. Now, we check whether $f_p \geq f_r$. If so, the pattern move might no longer be appropriate and we do an initial step (I). Otherwise, the pattern move and the exploration steps lead to an improvement and we do a basic iteration again (II).

(III) Step Size Reduction
The relative step size for the exploration moves is reduced according to

$$\Delta x \leftarrow c\,\Delta x \tag{6.33}$$

where $0 < c < 1$ is the constant step reduction factor. A common value for c is 0.5. x_c is considered as the minimum point and the algorithm stops if the step size has been reduced m_{max} times.

c) Keywords

To invoke the Hooke-Jeeves algorithm, the `Algorithm` section of the GenOpt command file must have the following form:

```
Algorithm{
   Main                    = HookeJeeves;
   StepReduction           = PositiveDouble;
   NumberOfStepReduction   = PositiveInteger;
}
```

The entries are defined as follows:

`Main` The name of the main algorithm.

`StepReduction` The step reduction factor, c in (6.33), where $0 < c < 1$. A common value is $c = 0.5$.

`NumberOfStepReduction` The number that specifies how many times a step reduction has to be done before a point is considered as being a minimum point. `NumberOfStepReduction` is equal to the parameter m_{max} in Fig. 6.4. (A common value is $m_{max} = 2$, but m_{max} depends on the step size and the required accuracy of x).

6.3.2 Simplex Method of Nelder and Mead with the Extension of O'Neill

The Simplex method of Nelder and Mead is based on a direct comparison of function values without using derivatives. It can be used to seek a solution of (5.2), or, by using GenOpt's scheme that takes box-constraints into account, to seek a solution of (5.3).

The algorithm superimposes an n-dimensional simplex in the space that is spanned by the free parameters ($n > 1$). At each of the $(n + 1)$ vertices of the simplex, the value of the objective function is evaluated. In each iteration step, the point with the highest value of the objective function is replaced by another point. The algorithm consists of three main operations: (a) *point reflection*, (b) *contraction of the simplex* and (c) *expansion of the simplex*.

It is known that the Simplex method may fail to converge to a stationary point, even if the objective function is smooth. See, for example, an excellent discussion in [Wri96], or [McK98, LRWW98]. The method fails when the simplex collapses into a subspace, or becomes extremely elongated and distorted in shape. Despite its bad convergence properties, the Simplex method often successfully locates a greatly improved solution with many fewer function evaluations than its competitors [Wri96].

In [McK98], McKinnon states a strictly convex function with three continuous derivatives and a set of initial iterates for which the Simplex method converges to a nonstationary point. In this example, the vertices tend to a straight line which is orthogonal to the steepest descent direction.

a) Main Operations

The notation defined below is used in describing the main operations. The operations are illustrated in Fig. 6.5 where, for simplicity, a two-dimensional simplex is considered.

Notation:

- $\mathbf{I}$ is the set of all vertex indices, i.e.,

$$\mathbf{I} \triangleq \{1, \ldots, n+1\}. \tag{6.34}$$

- h is a suffix that denotes the point with the highest function value:

$$h = \arg\max_{i \in \mathbf{I}} f(x_i), \tag{6.35}$$

  that is

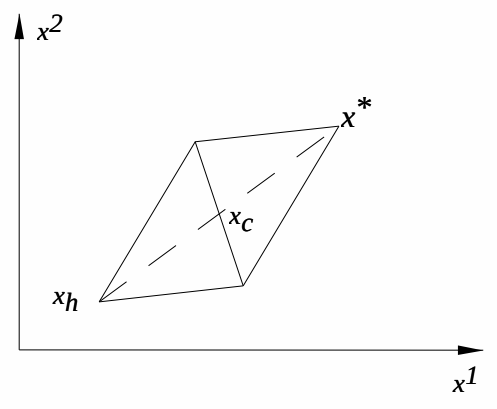
$$f_h(x) = \max_{i \in \mathbf{I}} f(x_i). \tag{6.36}$$

- l denotes the point with the lowest function value:

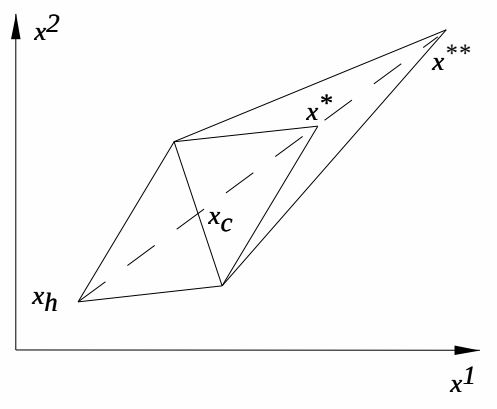
$$l = \arg\min_{i \in \mathbf{I}} f(x_i). \tag{6.37}$$

- For simplicity, we will often omit the argument of the function, i.e., we define the notation

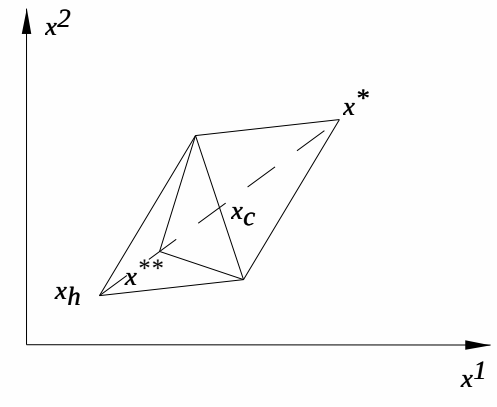
$$f_i \triangleq f_i(x). \tag{6.38}$$



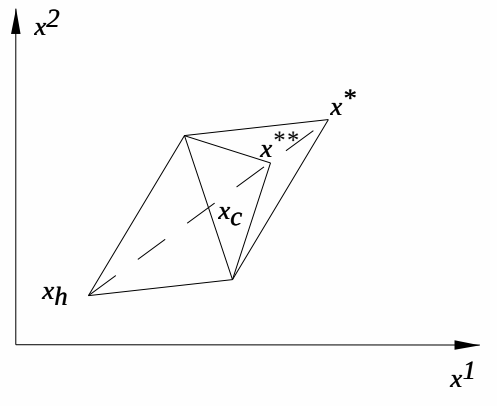
(a) Reflection



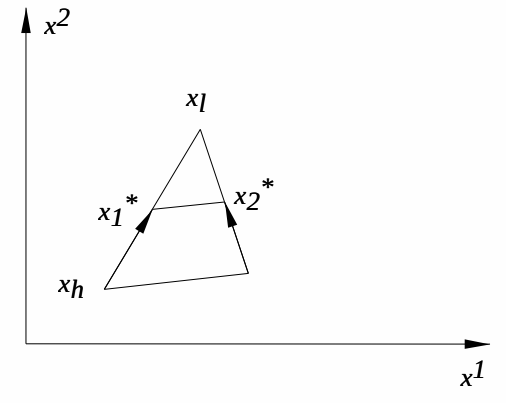
(b) Expansion



(c) Partial inside contraction



(d) Partial outside contraction



(e) Total contraction

Fig. 6.5: Simplex operations

- x_c is the centroid of the simplex, defined as

$$x_c \triangleq \frac{1}{n} \sum_{\substack{i=1 \\ i \neq h}}^{n+1} x_i \tag{6.39}$$

Using this notation, the three main operations are:

Reflection The reflection of $x_h \in \mathbb{R}^n$ to a point denoted as $x^* \in \mathbb{R}^n$ is defined as

$$x^* \triangleq (1+\alpha)\, x_c - \alpha\, x_h, \tag{6.40}$$

where $\alpha \in \mathbb{R}$ is a positive constant, called the *reflection coefficient*.

Expansion of the simplex The *expansion coefficient*, $\gamma \in \mathbb{R}$, is defined as

$$\gamma \triangleq \frac{\|x^{**} - x_c\|}{\|x^* - x_c\|} > 1. \tag{6.41}$$

The expansion of $x^* \in \mathbb{R}^n$ to $x^{**} \in \mathbb{R}^n$ is given by

$$x^{**} \triangleq \gamma\, x^* + (1-\gamma)\, x_c. \tag{6.42}$$

Contraction of the simplex The *contraction coefficient*, $\beta \in \mathbb{R}$, is defined as

$$0 < \beta \triangleq \frac{\|x^{**} - x_c\|}{\|x_h - x_c\|} < 1. \tag{6.43}$$

Hence, the new point, denoted by $x^{**} \in \mathbb{R}^n$, is given by

$$x^{**} \triangleq \beta\, x_h + (1-\beta)\, x_c. \tag{6.44}$$

.

b) Basic Algorithm

In this section, the basic Nelder and Mead algorithm is described [NM65]. The extension of O'Neill and the modified restart criterion are discussed later. The algorithm is as follows:

1. Initialization: Construct an initial simplex, spanned by $(n+1)$ points and calculate the function values at each vertex. Each vertex is defined as

$$x_{i+1} = x_1 + c\, s_i\, e_i, \qquad i \in \{1, 2, \ldots, n\}, \tag{6.45}$$

where s_i is the i-th component of the vector with the step size of the free parameters, e_i is the unity vector where the i-th component is one and all other are zero, and c is a scalar equal to one for the first initialization of the algorithm.

2. Reflection: The worst point, that is the point with the highest function value, is reflected at the centroid, which leads to x^*.

3. Check if we got the best point: If $f^* < f_l$, try to expand the simplex in the direction $x^* - x_c$ since further improvement in this direction is likely. If the expansion according to (6.42) is successful, that is $f^{**} < f_l$, then x^{**} is taken as the new vertex, otherwise x^* is the new vertex and the procedure is restarted from 2.

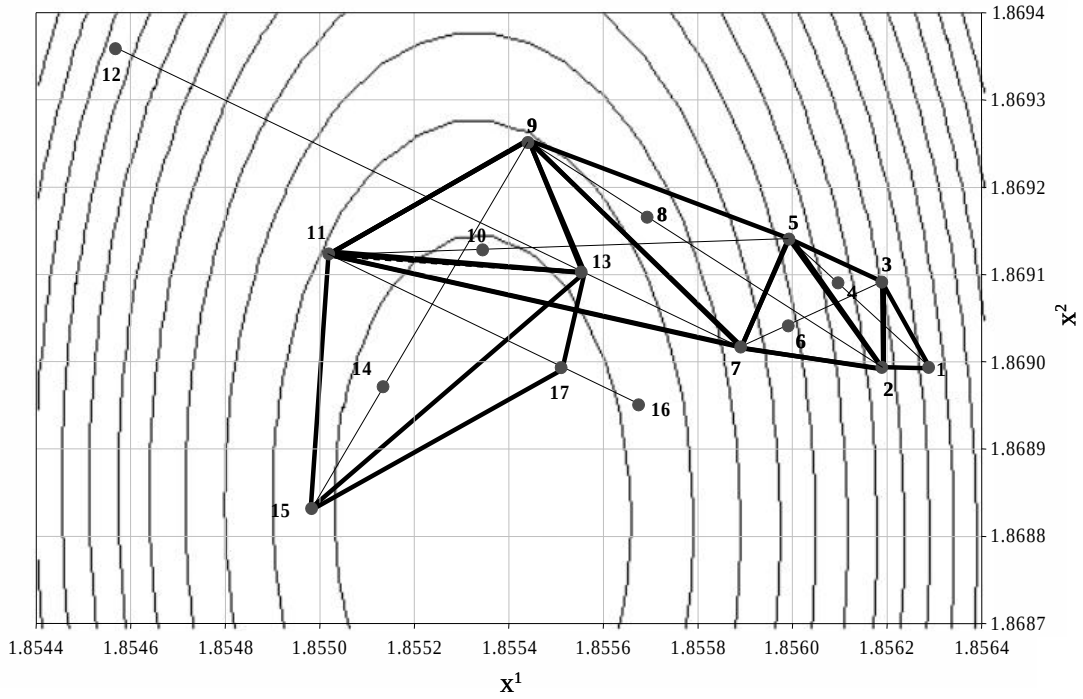


Fig. 6.6: Sequence followed by the Simplex algorithm during optimization

4. If it turned out under 3 that $f^* \geq f_l$, then we did not get the best point. So, we check if the new point x^* is the worst of all points. If $f^* > f_i$ for all $i \neq h$, we contract the simplex (see 5); otherwise x^* replaces x_h as the new vertex and the calculation is restarted from 2.

5. For the contraction, we first check if we should try a partial outside contraction or a partial inside contraction: If $f^* \geq f_h$, that is our new point x^* is as bad or even worse than x_h, then we try a partial inside contraction. To do so, we leave our indices as is and apply (6.44). Otherwise, we try a partial outside contraction. This can be simply done by replacing x_h by x^* and applying (6.44). After both cases – the partial inside or the partial outside contraction – we continue at 6.

6. We now check if our new point x^{**} is the worst point of the new simplex, that is $f^{**} \geq f_h$.[1] If so, we do a total contraction of the simplex by replacing all x_i by $x_i \leftarrow (x_i + x_l)/2$. Otherwise, we replace x_h by x^{**}. In both cases we continue from 2.

Fig. 6.6 shows an example sequence of the optimization process. The sequence starts with constructing an initial simplex x_1, x_2, x_3. x_1 has the

[1] Nelder and Mead [NM65] use the strict inequality $f^{**} > f_h$. However, if the user writes the objective function value only with a few representative digits to a text file, then the function looks like a step function if slow convergence is achieved. In such cases, f^{**} might sometimes be equal to f_h. Experimentally, it has been shown advantageous to perform, then, a total contraction rather than continuing with a reflection. Therefore, the strict inequality has been changed to a weak inequality.

highest function value and is therefore reflected, which leads to x_4. x_4 is the best point in the set $\{x_1, x_2, x_3, x_4\}$. Thus, it is further expanded, which gives x_5. x_2, x_3 and x_5 now span the new simplex. In this simplex, x_3 is the vertex with the highest function and hence goes over to x_6 and further to x_7. The process of reflection and expansion is continued again two times, which leads to the simplex spanned by x_7, x_9 and x_{11}. x_7 goes over to x_{12} which turns out to be the worst point. Hence, we do a partial inside contraction, which gives x_{13}. x_{13} is better than x_7 so we use the simplex spanned by x_9, x_{11} and x_{13} for the next reflection. The last steps of the optimization are for clarity not shown in Fig. 6.6.

c) Stopping Criteria

The first criterion is a test of the variance of the function values at the vertices of the simplex:

$$\mathrm{var}f = \frac{1}{n}\left(\sum_{i=1}^{n+1}\big(f(x_i)\big)^2 - \frac{1}{n+1}\left(\sum_{i=1}^{n+1} f(x_i)\right)^2\right) < \epsilon^2. \tag{6.46}$$

If $\mathrm{var}f$ is less than the square of a prescribed value ϵ, then the original implementation of the algorithm would stop. Nelder and Mead have chosen this stopping criterion based on statistical problems related to finding the minimum of a sum-of-squares surface. In these problems the curvature near the minimum gives information about the unknown parameters. A slight curvature indicates a high sampling variance of the estimate and therefore there is no reason for finding the minimum point with high accuracy. However, if the curvature is marked, then the sampling variance is low and a higher accuracy in determining the optimal parameter set is desirable.

d) O'Neill's Modification

O'Neill modified the termination criterion by adding a further condition [O'N71]. He checks whether any orthogonal step, each starting from the best vertex of the current simplex, leads to a further improvement of the objective function. He therefore sets $c = 0.001$ and tests if the *optimality condition*

$$f(x_l) < f(x) \tag{6.47a}$$

holds for all x defined by

$$x = x_l + c\, s_i\, e_i, \qquad i \in \{1, 2, \ldots, n\}, \tag{6.47b}$$

whereas x_l denotes the best known point, and s_i and e_i are defined as in (6.45).

e) Modification of Stopping Criteria

In GenOpt, (6.47) has been modified. It has been observed that users sometimes write the objective function value only with few representative digits to the output file. In such cases, (6.47a) won't be satisfied if the write statement in the simulation program truncates the value so that the difference $f(x_l) - f(x)$, where $f(\cdot)$ denotes the value that is found in the output file, is zero. To overcome this numerical problem (6.47b) has been modified to

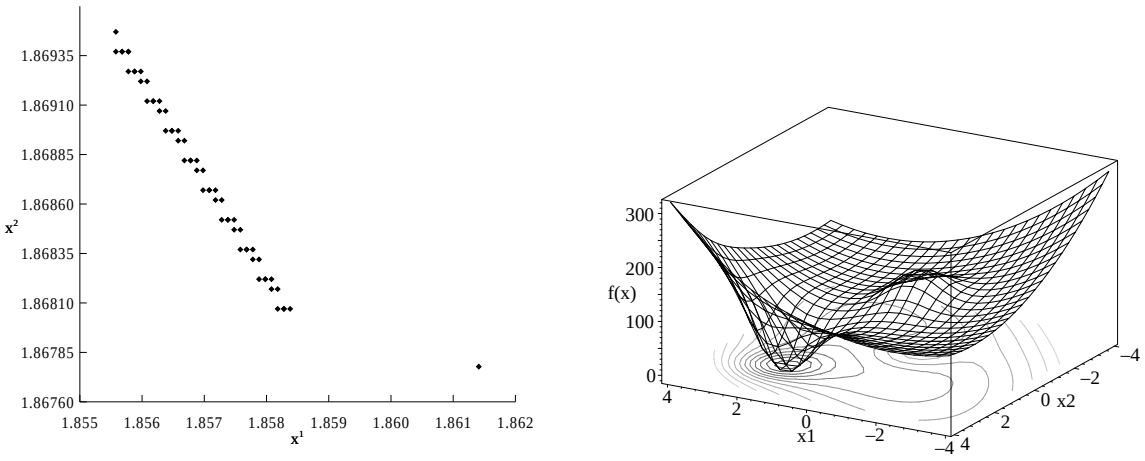
$$x = x_l + \exp(j)\, c\, s_i\, e_i, \qquad i \in \{1, 2, \ldots, n\} \tag{6.47c}$$

where for each direction i, the counter j is set to zero for the first trial and increased by one as long as $f(x_l) - f(x) = 0$. If (6.47a) fails for any direction, then x according to (6.47c) is the new starting point and a new simplex with side length $(c\, s_i)$ is constructed. The best currently known point, that is the point x that failed (6.47a), is used as initial point, x_l, in (6.45).

Tests on several functions showed that during slow convergence the routine was restarted too often.

Fig. 6.7(a) shows a sequence of points in such an optimization phase where the routine was restarted too often. The points shown are a part of the iteration sequence in an area close to the minimum of the test function shown in Fig. 6.7(b). The algorithm reaches the neighborhood of the minimum with appropriately large steps. The last of these steps can be seen at the right in Fig. 6.7(a). After this step, the stopping criterion (6.46) was satisfied which led to a restart check, followed by a new construction of the simplex. From there on, the convergence was very slow due to the small step size. After each step, the stopping criterion was satisfied again which led to a new test of the optimality condition (6.47a), followed by a reconstruction of the simplex. This check is very costly in terms of function evaluations and, furthermore, the restart with a new simplex does not allow increasing the step size, though we are heading (locally) in the right direction.

O'Neill's modification prevents both excessive checking of the optimality condition as well as excessive reconstruction of the initial simplex. This is done by checking for convergence only after a predetermined number of steps (i.e., after five iterations). However, the rate of convergence of the algorithm depends crucially on this number. As an extreme case, a few test runs were done



(a) Sequence of generated points in the neighborhood of the function minimum

(b) 2-dimensional test function "2D1"

Fig. 6.7: Nelder Mead Trajectory

where convergence was checked after each step as in Fig. 6.7(a). It turned out that in some cases no convergence was reached within a moderate number of function evaluations if ϵ in (6.46) is chosen to large, i.e., $\epsilon = 10^{-3}$ (see Tab. 6.1).

To make the algorithm more robust, it is modified based on the following arguments:

1. if the simplex is moving in the same direction in the last two steps, then the search should not be interrupted by checking for optimality since we are making steady progress in the moving direction.
2. if we do *not* have a partial inside or total contraction immediately beyond us, then it is likely that the minimum lies in the direction currently being explored. Hence, we do not want to interrupt the current search with a possible restart.

These considerations have led to two criteria that both have to be satisfied to permit the convergence check according to (6.46), which might be followed by a check for optimality.

First, it is checked if we have done a partial inside contraction or a total contraction. If so, we check if the direction of the latest two steps in which the simplex is moving has changed by an angle of at least $(\pi/2)$. To do so, we introduce the center of the simplex, defined by

$$x_m \triangleq \frac{1}{n+1} \sum_{i=1}^{n+1} x_i, \tag{6.48}$$

and the normalized direction of the simplex between two steps,

$$d_k \triangleq \frac{x_{m,k+1} - x_{m,k}}{\|x_{m,k+1} - x_{m,k}\|_2}. \tag{6.49}$$

	Accuracy							
	$\epsilon = 10^{-3}$				$\epsilon = 10^{-5}$			
Test function	Rosenbrock	2D1	Quad with I matrix	Quad with Q matrix	Rosenbrock	2D1	Quad with I matrix	Quad with Q matrix
Original (with reconstruction)	137	120	3061	1075	139	109	1066	1165
Original, but no reconstruction	136	110	1436	1356	139	109	1433	1253
Modified, with reconstruction	145	112	1296	1015	152	111	1060	1185
Modified, no reconstruction	155	120	1371	1347	152	109	1359	1312

Tab. 6.1: Comparison of the number of function evaluations for different implementations of the simplex algorithm. See Appendix for the definition of the function

Now, we can determine how much the simplex has changed its direction d_k between two steps by looking at the inner product of d_{k-1} and d_k. The inner product is equal to the cosine of the angle between the moving direction. If it is not larger than zero, i.e.,

$$\cos \phi_k = \langle d_{k-1},\, d_k \rangle \leq 0 \tag{6.50}$$

then the moving direction of the simplex has changed by at least $\pi/2$ and so we have changed our exploration direction. If this is the case, we might be at the minimum and hence we test the variance of the vertices (6.46), possibly followed by a test of the optimality condition (6.47a).

Besides the above modification, a further modification was tested: In some cases a reconstruction of the simplex after a failed convergence check – (6.47a) does not hold for all proven directions – seemed to slow down convergence. Therefore, the algorithm was modified so that it continues at point 2 on page 26 without reconstructing the simplex after failing the convergence check. However, reconstructing the simplex led in most of the benchmark tests to faster convergence. Therefore, this modification is no longer used in the algorithm.

f) Benchmark Tests

Tab. 6.1 shows the number of function evaluations and Fig. 6.8 shows the relative number of function evaluations compared to the original implementation for several test cases. The different functions and the parameter

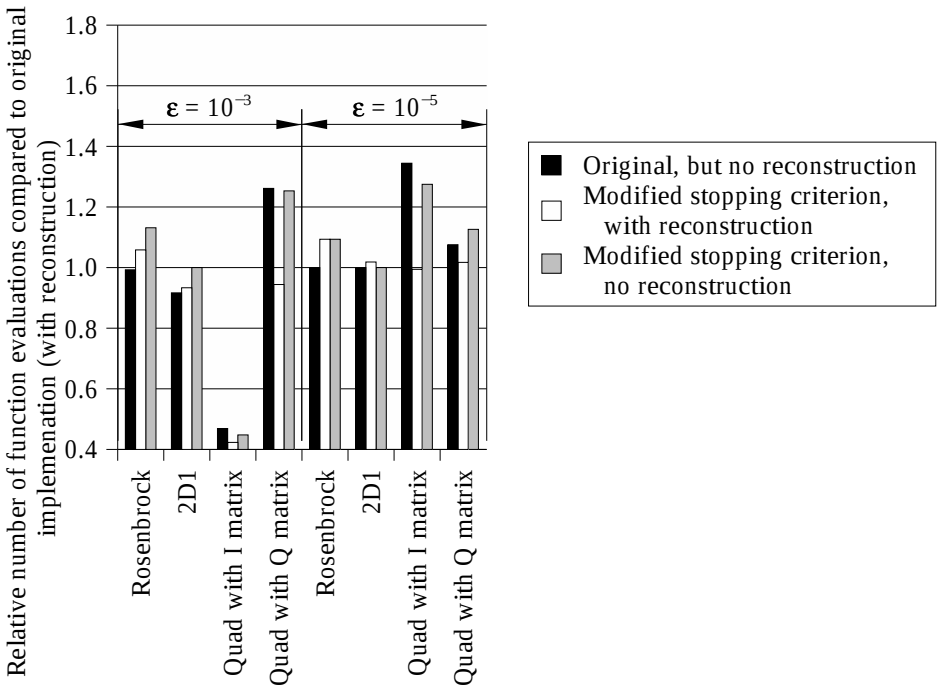


Fig. 6.8: Comparison of benchmark tests

settings are given in the Appendix. The only parameter that was changed for the different optimizations is the accuracy, ϵ. Furthermore, in some cases the modification of the stopping criterion and/or the reconstruction of a new simplex is implemented in the algorithm.

It turned out that modifying the stopping criterion is effective in most cases, particularly if a new simplex is constructed after the check for optimality failed. Therefore, the following version of the simplex algorithm was finally implemented in GenOpt:

1. The base algorithm of Nelder and Mead, including the extension of O'Neill, can be applied. That means, that after failing the optimality check, the simplex is *always* reconstructed with the new step size.
2. The user can choose if the stopping criterion should be modified as explained in the previous chapter (which means inner contraction *and* moving direction of the simplex have to be satisfied).

g) Keywords

To invoke the Simplex algorithm, the `Algorithm` section of the GenOpt command file must have following form:

```
Algorithm{
   Main                    = NelderMeadONeill;
   Accuracy                = PositiveDouble;
   StepSizeFactor          = PositiveDouble;
   BlockRestartCheck       = PositiveInteger;
   ModifyStoppingCriterion = boolean;
}
```

The key words have following meaning:

`Main` The name of the main algorithm.

`Accuracy` The accuracy that has to be reached before the optimality condition is checked. `Accuracy` is defined as equal to ϵ of (6.46), page 28.

`StepSizeFactor` A factor that multiplies the step size of each parameter for (a) testing the optimality condition and (b) reconstructing the simplex. `StepSizeFactor` corresponds to c in (6.45) and (6.47c).

`BlockRestartCheck` Number that indicates for how many main iterations the restart criterion is not checked. If zero, restart might be checked after each main iteration.

`ModifyStoppingCriterion` Flag indicating whether the stopping criterion should be modified. If `true`, the optimality check (6.46) is done only if both of the following conditions are satisfied: (a) in the last step, either a partial or a total inner contraction was done, and (b) the moving direction of the simplex has changed by an angle of at least $(\pi/2)$, whereas the direction is computed using (6.50).

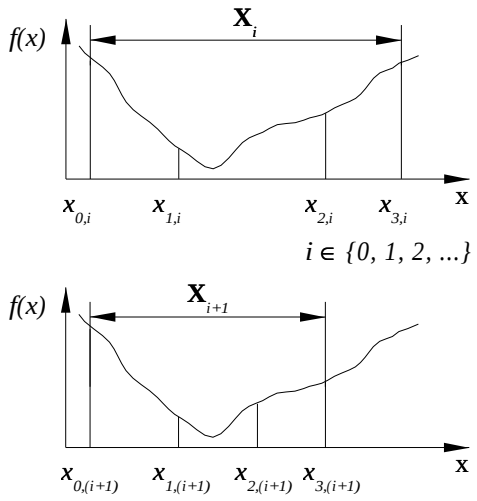


Fig. 6.9: Interval division

6.3.3 Interval Division Methods

The implementation of the interval division methods is explained first for a master algorithm for one-dimensional minimization. The algorithm does not require derivatives and it requires only one function evaluation per interval division, except for the initialization. It can be used to minimize a one-dimensional function, $f\colon \mathbb{R} \rightarrow \mathbb{R}$, in a given *unimodal* interval.

The master algorithm is used to implement two commonly used interval division methods: the golden section search and the Fibonacci division. First, they will be explained in the general framework of the master algorithm since they differ only by the ratio of the interval division.

Even though the described algorithms do a one-dimensional minimization, they are explained for the case where $f\colon \mathbb{R}^n \rightarrow \mathbb{R}$, whereas all $x \in \mathbb{R}^n$ must lie on the line connecting the bounds of the unimodal interval. This is done since the implementation of the algorithms is such that they can be used either for minimization of $f\colon \mathbb{R} \rightarrow \mathbb{R}$, or for doing a line-search that is used by multi-dimensional algorithms.

a) General Interval Division

Let $\mathbf{X}$ be an interval such that

$$\mathbf{X} = \{x \in \mathbb{R}^n \mid x = x_0 + s\,(x_3 - x_0)\,;\ s \in [0,\,1]\} \tag{6.51}$$

on which $f\colon \mathbb{R} \rightarrow \mathbb{R}$ is unimodal, i.e., there exists one unique point, x^*, such that $f(x) - f(x^*) > 0$ for all $x \in \mathbf{X} \setminus x^*$.[2] Thus, x^* is the unique minimizer of $f(\cdot)$ in $\mathbf{X}$. Then we can evaluate two points, x_1 and x_2, such that

$$x_1 = x_0 + s\,(x_3 - x_0), \qquad s \in (0,\,1), \tag{6.52}$$

[2] The notation $x \in \mathbf{X} \setminus x^*$ means x is an element of the set $\mathbf{X}$ excluding the point x^*

and

$$x_2 = x_1 + s\,(x_3 - x_1), \qquad s \in (0,\, 1). \tag{6.53}$$

Since $f(\cdot)$ is unimodal by assumption, at least one of the inequalities

$$f(x_1) \;\leq\; \min\{f(x_0),\, f(x_3)\}, \tag{6.54}$$
$$f(x_2) \;\leq\; \min\{f(x_0),\, f(x_3)\}, \tag{6.55}$$

must hold. (In fact, both inequalities should be strict due to unimodality, but because of finite precision of computers, we allow weak inequalities.)

Suppose that (6.54) holds.
Case 1: Suppose that $f(x_1) \leq f(x_2)$. Then we know by unimodality that $x^* \in [x_0,\, x_2]$. Hence, we can discard the interval $(x_2,\, x_3]$ and restrict our search to $[x_0,\, x_2]$.
Case 2: We must have $f(x_1) > f(x_2)$. Again by unimodality, we have $x^* \in [x_1,\, x_3]$ and we can discard $[x_0,\, x_1)$.

Similar conclusions hold if (6.55) is satisfied. In all cases we have succeeded in reducing the initial interval to a new interval that contains the minimizer, x^*.

By constructing the new set $\{x_{k,i}\}_{k=0}^{3}$, we want to nest the sequence of intervals

$$[x_{0,(i+1)},\, x_{3,(i+1)}] \subset [x_{0,i},\, x_{3,i}], \qquad i \in \{0,\, 1,\, 2,\, \ldots\}, \tag{6.56}$$

so that we have to evaluate $f(\cdot)$ in each step at one new point only. To do so, we can assign the new bounds of the interval so that either $[x_{0,(i+1)},\, x_{3,(i+1)}] = [x_{0,i},\, x_{2,i}]$, or $[x_{0,(i+1)},\, x_{3,(i+1)}] = [x_{1,i},\, x_{3,i}]$, depending on which interval has to be discarded. By doing so, we have to evaluate only one new point in the interval. It remains to decide where to locate the new point, in which the two algorithms differ.

b) Golden Section Interval Division

Suppose we have three points in an interval so that

$$x_1 = x_0 + s\,(x_3 - x_0), \qquad s \in (0,\, 1), \tag{6.57}$$

and

$$\frac{\|x_0 - x_1\|}{\|x_0 - x_3\|} = q \tag{6.58}$$

and, hence,

$$\frac{\|x_1 - x_3\|}{\|x_0 - x_3\|} = 1 - q. \tag{6.59}$$

Suppose that our new point being located is a fraction, say w, beyond x_1, that is

$$\frac{\|x_1 - x_2\|}{\|x_0 - x_3\|} = w. \tag{6.60}$$

Now, the new segment will either be of length $q + w$, or $1 - q$, relative to the current one. To reduce the worst possible case, we have to choose w such that those two intervals are of the same length, namely

$$q + w = 1 - q. \tag{6.61}$$

Since the new point, x_2, is, due to construction, symmetric to x_1 in the interval $[x_0,\, x_3]$, we know that it must be positioned in the larger of the intervals $[x_0,\, x_1]$ and $[x_1,\, x_3]$. We still have to determine the fraction q. Since we apply the process of interval division recursively, we now by scale similarity that

$$\frac{w}{1-q} = q \tag{6.62}$$

must hold. Combining (6.61) and (6.62) leads to

$$q^2 - 3q + 1 = 0, \tag{6.63}$$

with solutions

$$q_{1,2} = \frac{3 \pm \sqrt{5}}{2}. \tag{6.64}$$

Since $q < 1$ by (6.58), the solution of interest is

$$q = \frac{3 - \sqrt{5}}{2} \approx 0.382. \tag{6.65}$$

The fractional distances $q \approx 0.382$ and $1 - q \approx 0.618$ correspond to the so-called *golden section*, which gives this method its name.

Note that the interval is reduced in each step by the fraction $1 - q$, i.e., we have *linear convergence*. In the m-th iteration, we have

$$\begin{aligned} \|x_{0,\,m} - x_{2,\,m}\| &= \|x_{1,\,m} - x_{3,\,m}\| = \|x_{0,\,(m+1)} - x_{3,\,(m+1)}\| \\ &= (1-q)^{m+1}\, \|x_{0,\,0} - x_{3,\,0}\|. \end{aligned} \tag{6.66}$$

Hence, the required number of iterations, m, to reduce the initial interval of uncertainty $\|x_{0,\,0} - x_{3,\,0}\|$ to at least a fraction, say r, defined by

$$r \triangleq \frac{\|x_{0,\,m} - x_{2,\,m}\|}{\|x_{0,\,0} - x_{3,\,0}\|} = \frac{\|x_{1,\,m} - x_{3,\,m}\|}{\|x_{0,\,0} - x_{3,\,0}\|} \tag{6.67}$$

is given by

$$m \geq \frac{\ln r}{\ln(1-q)} - 1. \tag{6.68}$$

c) Fibonacci Division

Another way to divide an interval such that we need one function evaluation per iteration can be constructed as follows: Given an initial interval $[x_{0,\,i}, x_{3,\,i}]$, $i = 0$, we divide it into three segments symmetrically around its midpoint. Let $d_{1,\,i} < d_{2,\,i} < d_{3,\,i}$ denote the distance of the segment endpoints, measured from $x_{0,\,i}$. Then we have by symmetry $d_{3,\,i} = d_{1,\,i} + d_{2,\,i}$. By the bracket elimination procedure explained above, we know that we are discarding a segment of length $d_{1,\,i}$. Therefore, our new interval is of length $d_{3,\,(i+1)} = d_{2,\,i}$. By symmetry we also have $d_{3,\,(i+1)} = d_{1,\,(i+1)} + d_{2,\,(i+1)}$. Hence, if we construct our segment

length such that $d_{3,(i+1)} = d_{1,(i+1)} + d_{2,(i+1)} = d_{2,i}$ we can reuse one known point. Such a construction can be done by using *Fibonacci* numbers, which are defined recursively by

$$F_0 \triangleq F_1 \triangleq 1, \tag{6.69}$$
$$F_i \triangleq F_{i-1} + F_{i-2}, \qquad i \in \{2, 3, \ldots\}. \tag{6.70}$$

The first few numbers of the Fibonacci sequence are $\{1, 1, 2, 3, 5, 8, 13, 21, \ldots\}$. The length of the intervals $d_{1,i}$ and $d_{2,i}$, respectively, are then given by

$$d_{1,i} = \frac{F_{m-i}}{F_{m-i+2}}, \quad d_{2,i} = \frac{F_{m-i+1}}{F_{m-i+2}}, \qquad i \in \{0, 1, \ldots, m\}, \tag{6.71}$$

where $m > 0$ describes how many iterations will be done. Note that m must be known prior to the first interval division. Hence, the algorithm must be stopped after a predetermined number of iterations, m.

The reduction of the length of the uncertainty interval per iteration is given by

$$\frac{d_{3,(i+1)}}{d_{3,i}} = \frac{d_{2,i}}{d_{1,i} + d_{2,i}} = \frac{\frac{F_{m-i+1}}{F_{m-i+2}}}{\frac{F_{m-i}}{F_{m-i+2}} + \frac{F_{m-i+1}}{F_{m-i+2}}} = \frac{F_{m-i+1}}{F_{m-i+2}}. \tag{6.72}$$

Hence, after m iterations we have

$$\begin{aligned} \frac{d_{3,m}}{d_{3,0}} &= \frac{d_{3,m}}{d_{3,(m-1)}} \frac{d_{3,(m-1)}}{d_{3,(m-2)}} \cdots \frac{d_{3,2}}{d_{3,1}} \frac{d_{3,1}}{d_{3,0}} \\ &= \frac{F_2}{F_3} \frac{F_3}{F_4} \cdots \frac{F_m}{F_{m+1}} \frac{F_{m+1}}{F_{m+2}} = \frac{2}{F_{m+2}}. \end{aligned} \tag{6.73}$$

The required number of iterations, m, to reduce the initial interval, $d_{3,0}$, to at least a fraction, r, defined by (6.67), can again be obtained by expansion from

$$\begin{aligned} r &= \frac{d_{2,m}}{d_{3,0}} = \frac{d_{3,(m+1)}}{d_{3,0}} = \frac{d_{3,(m+1)}}{d_{3,m}} \frac{d_{3,m}}{d_{3,(m-1)}} \cdots \frac{d_{3,2}}{d_{3,1}} \frac{d_{3,1}}{d_{3,0}} \\ &= \frac{F_1}{F_2} \frac{F_2}{F_3} \cdots \frac{F_m}{F_{m+1}} \frac{F_{m+1}}{F_{m+2}} = \frac{1}{F_{m+2}}. \end{aligned} \tag{6.74}$$

Hence, m is given by

$$m = \arg\min \left\{ m \mid r \geq \frac{1}{F_{m+2}} \right\}. \tag{6.75}$$

d) Comparison of Efficiency

The golden section is more efficient than the Fibonacci division. Comparing the reduction of the interval of uncertainty, $\|x_{0,m} - x_{3,m}\|$, in the limiting case for $m \to \infty$, we obtain

$$\lim_{m\to\infty} \frac{\|x_{0,m} - x_{3,m}\|_{GS}}{\|x_{0,m} - x_{3,m}\|_F} = \lim_{m\to\infty} \frac{F_{m+2}}{2} (1-q)^m = 0.95. \tag{6.76}$$

e) Master Algorithm for Interval Division

The following algorithm explains the steps of the interval division method.

Data: x_0, x_3;
Procedure that returns r_i, defined as
$r_i = \|x_{0,i} - x_{2,i}\| / \|x_{0,0} - x_{3,0}\|$
Step 1: Initialize
$\Delta x = x_3 - x_0$;
$x_2 = x_0 + r_1 \Delta x$;
$x_1 = x_0 + r_2 \Delta x$;
$f_1 = f(x_1)$; $f_2 = f(x_2)$;
$i = 2$;
Step 2: Iterate
$i = i + 1$;
if $(f_2 < f_1)$
$x_0 = x_1$; $x_1 = x_2$;
$f_1 = f_2$;
$x_2 = x_3 - r_i \Delta x$;
$f_2 = f(x_2)$;
else
$x_3 = x_2$; $x_2 = x_1$;
$f_2 = f_1$;
$x_1 = x_0 + r_i \Delta x$;
$f_1 = f(x_1)$;
Step 3: stop or go to Step 2;

f) Keywords

To invoke the golden section or the Fibonacci Division algorithm, the `Algorithm` section of the GenOpt command file must have following form:

```
Algorithm{
   Main              = GoldenSection | Fibonacci;
  [AbsDiffFunction   = PositiveDouble;  |
   IntervalReduction = PositiveDouble;  ]
}
```

The keywords have following meaning

`Main` The name of the main algorithm.

The following two keywords are optional. If none of them is specified, then the algorithm stops after `MaxIte` function evaluations (i.e., after `MaxIte`−2 iterations), where `MaxIte` is specified in the section `OptimizationSettings`. If both of them are specified, an error occurs.

`AbsDiffFunction` The absolute difference defined as

$$\Delta f \triangleq |\min\{f(x_0),\, f(x_3)\} - \min\{f(x_1),\, f(x_2)\}|. \tag{6.77}$$

If Δf is lower than `AbsDiffFunction`, the search stops successfully.
Note: Since the maximum number of interval reductions must be known for the initialization of the Fibonacci algorithm, this keyword can be used only for the golden section algorithm. It must not be specified for the Fibonacci algorithm.

`IntervalReduction` The required maximum fraction, r, of the end interval length relative to the initial interval length (6.67).

6.3.4 Parametric Runs

The `EquMesh` algorithm allows making parametric runs on an orthogonal, equidistant grid that is spanned in the space of the design parameters. To do so, each parameter must have a lower and upper bound (i.e., the keywords `Min` and `Max` of the Parameter section must be specified). The value of `Step` (which must be an integer greater than or equal to zero) specifies into how many intervals each axis will be divided. Thus, the setting

```
Vary{
   Parameter{ Name = x0; Min = -1; Ini = 0; Max = 1; Step = 1; }
   Parameter{ Name = x1; Min = -1; Ini = 0; Max = 1; Step = 2; }
}
```

would evaluate the objective function at all points $x \in \mathbf{X}$, where the set$\mathbf{X}$ is $\mathbf{X} \triangleq \{[-1, -1],[1, -1], [-1, 0], [1, 0], [-1, 1], [1, 1]\}$.

If the value of `Step` is equal to zero, then this parameter is fixed at the value specified by `Min`.

Note that the number of function evaluations increases exponentially with the number of free parameters, e.g., a 5-dimensional grid with 2 intervals in each dimension requires $3^5 = 243$ function evaluations, whereas a 10-dimensional grid would require $3^{10} = 59049$ function evaluations.

The `EquMesh` algorithm is invoked by the following specification in the command file:

```
Algorithm{
   Main = EquMesh;
}
```

Note that the whole section `OptimizationSettings` of the command file is ignored.

6.3.5 Choice of Algorithm

As stated above, there is no general optimization algorithm that works efficiently on all problems. The following section gives some guidelines on selecting an algorithm to solve a particular problem. Since in GenOpt the objective function is evaluated as a black-box function, the following guide assumes that no analytical information is known about the objective function. Also, the hints are given for problems with only a small number of free parameters and not for large-scale problems with hundreds or even thousands of free parameters.

The main questions to ask in selecting an efficient algorithm are:

1. Is the objective function continuously differentiable?
2. Is the objective function convex?
3. Is the number of free parameters high?

4. Are we doing parameter fitting?

5. Can the objective function be well approximated by a quadratic?

If the objective function is continuously differentiable, then, for some optimization algorithms, convergence to a stationary point can be proven. Lack of differentiability makes it often impossible to state any convergence criteria.

If the objective function or if the feasible domain of the free parameters is not convex, then the optimization algorithm may converge to a local minimizer rather than a global minimizer. There are algorithms that increase the probability of finding the global minimum but they do not guarantee finding it. If the number of local minima is small it may be worthwhile using any of the optimization algorithms described above and beginning the optimization with different starting values or changing the algorithm to increase the chance of finding the global minimum. However, bear in mind that finding a local minimum is still a better solution than doing no optimization at all.

It is obvious that the gradient-based methods do a poor job if the gradient does not provide representative information about the behavior of the function. Hence, if the function looks like case Fig. 5.2(a) on page 10 gradient-based methods should be avoided.

The Newton method should not be used by itself since it has no sense whether it is heading towards a maximum or a minimum. Furthermore, it requires the evaluation of the Hessian matrix, which is very costly. Also, the steepest descent method is not a good choice since it is very inefficient because of the orthogonal steps that it generates. However, both serve as good bases for developing algorithms and so they are used as a framework for other algorithms.

The BFGS algorithm and the modified Powell algorithm might be appropriate if the objective function can be assumed to be quadratic. Because of the conjugate search direction they work particularly well in this case. The modified Powell algorithm is good for data fitting [Wal75, p. 139] but should not be used if the number of free parameters exceeds 10 to 20.

If the objective function is continuously differentiable, it may be worthwhile spending the effort to approximate the gradient for the BFGS method. The BFGS method should be favored over the DFP method since it is less sensitive to numerical errors. The Fletcher-Reeves algorithm is not very advantageous for cases where the number of free parameters is small.

The Hooke-Jeeves method moves efficiently along the valley of the objective function, thereby reducing the dimensionality of the problem. Since it does not require explicit derivative information, it might be a good choice if the objective function is expected to have some discontinuities. If the objective function is continuously differentiable and has bounded level sets, then the Hooke-Jeeves algorithm converges to a stationary point [AD00].

The simplex method of Nelder-Mead is widely used. The method is very robust and not sensitive to numerical errors like the conjugate gradient methods are. Good convergence is often achieved if the number of free parameters is less than about 10.

7 Constraints

In most optimizations it is necessary to impose constraints on the free parameters and/or the dependent variables. An example of a constraint on a dependent variable is the case of minimizing heating energy by varying some parameters, such as the mass flow of the heating system. Without constraints minimum energy consumption would be achieved if the mass flow is zero, and hence the heating system does not run at all. To overcome this problem, we have to impose a constraint on a dependent variable. One possibility is to simply add a "penalty" term to the energy consumption. This could be such that every time the thermal comfort (one of our dependent variables) is not satisfied, a positive number is added to the energy consumption. If this number is chosen large enough, we can determine the optimal set of free parameters that minimizes the energy consumption while still ensuring thermal comfort.

In most optimization problems special techniques are required to take constraints into account. For example the objective function can be modified or the free parameters can be constrained. The following sections describe some ways of applying constraints.

The method used in GenOpt is described in Section 7.1.1. Since the penalty and barrier methods, which are described in Section 7.2, involve modifying the objective function, they are problem specific and have to be applied by the user.

7.1 Constraints on Free Parameters

7.1.1 Box Constraints

Box constraints are constant inequality constraints of the form

$$l^i \leq x^i \leq u^i, \qquad i \in \{1, 2, \dots, n\}, \tag{7.1}$$

where $l, x, u \in \mathbb{R}^n$ and $l^i < u^i$ for all $i \in \{1, 2, \dots, n\}$.

It is tempting to simply set the value x^i back to either l^i or u^i if the optimization algorithm puts x^i outside the $[l^i, u^i]$ range. However, this can lead to numerical difficulties.

A better approach is to transform the restricted parameters into another space where no restrictions have to be imposed, that is

$$x^i \leftrightarrow t^i, \qquad x^i \in [l^i, u^i], \quad t^i \in (-\infty, +\infty). \tag{7.2}$$

Of course, the new variable t^i must still be bounded to prevent an overflow. However, if the solution x^* of the optimization problem is bounded, then the transformed parameter set is also bounded.

Instead of optimizing the constrained parameter set, $x \in \mathbb{R}^n$, we now optimize the new parameter set, $t \in \mathbb{R}^n$, which is unconstrained. The transformation ensures that all parameters stay feasible during the whole iteration process. Furthermore, we will ensure that the i-th gradient component vanishes at l^i and u^i. Hence, the algorithm does not tend to cross those boundaries even if

the minimum lies in an unfeasible area.

Various equations are possible to perform the transformation. The transformations that are implemented in GenOpt are the following:

$$l^i \le x^i: \qquad t^i = \sqrt{x^i - l^i} \tag{7.3}$$
$$x^i = l^i + (t^i)^2 \tag{7.4}$$
$$l^i \le x^i \le u^i: \qquad t^i = \arcsin\left(\sqrt{\frac{x^i - l^i}{u^i - l^i}}\right) \tag{7.5}$$
$$x^i = l^i + (u^i - l^i)\,\sin^2 t^i \tag{7.6}$$
$$x^i \le u^i: \qquad t^i = \sqrt{u^i - x^i} \tag{7.7}$$
$$x^i = u^i - (t^i)^2 \tag{7.8}$$

7.1.2 Coupled Linear Constraints

In some cases the constraints have to be formulated in terms of a linear system of equations of the form

$$A\,x = b, \tag{7.9}$$

where $A \in \mathbb{R}^m \times \mathbb{R}^n$, $x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, and $\mathrm{rank}(A) = m$.

There are various algorithms that take this kind of restriction into account. However, such restrictions are rare in building simulation and thus not implemented in GenOpt. If there is nevertheless a need to impose such restrictions, they can be included by adding an appropriate optimization algorithm and retrieving the coefficients by using the methods offered in GenOpt's class `Optimizer`.

7.2 Constraints on Dependent Variables

Often the constraints are non-linear of the form

$$g(x) \le 0 \tag{7.10}$$
$$h(x) = 0 \tag{7.11}$$

where $g\colon \mathbb{R}^n \to \mathbb{R}^k$ and $h\colon \mathbb{R}^n \to \mathbb{R}^l$ are non-linear functions, and $x \in \mathbb{R}^n$.

This kind of restriction is often taken into account by adding *penalty* or *barrier* functions to the objective function. Let $T_g(\cdot)$ and $T_h(\cdot)$ be the vectors of transformation functions acting on the corresponding $g^i(x)$ and $h^i(x)$, i.e.,

$$T_g\big(g(x)\big) = \Big(T_g^1\big(g^1(x)\big),\quad T_g^2\big(g^2(x)\big),\quad \ldots\quad ,\quad T_g^k\big(g^k(x)\big)\Big), \tag{7.12}$$
$$T_h\big(h(x)\big) = \Big(T_h^1\big(h^1(x)\big),\quad T_h^2\big(h^2(x)\big),\quad \ldots\quad ,\quad T_h^l\big(h^l(x)\big)\Big), \tag{7.13}$$

with $T_g^i\colon \mathbb{R} \to \mathbb{R}$ and $T_h^i\colon \mathbb{R} \to \mathbb{R}$ for all i.

Let $w_g \in \mathbb{R}^k$ and $w_h \in \mathbb{R}^l$ denote the sets of positive weighting constants for $g \in \mathbb{R}^k$ and $h \in \mathbb{R}^l$, respectively. Using this notation, we can modify the objective function to the form

$$\hat{f}(x, w_g, w_h) \triangleq f(x) + \int_D \Big(\big\langle w_g,\, T_g\big(g(x)\big)\big\rangle + \big\langle w_h,\, T_h\big(h(x)\big)\big\rangle\Big) dD. \tag{7.14}$$

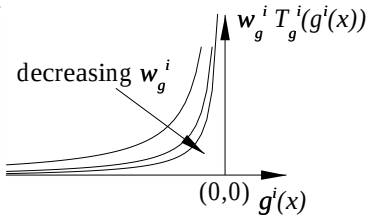


Fig. 7.1: Barrier function

In (7.14), the integral indicates that violations of the constraints can be added to the objective function value $f(x)$ during any stage of the simulation. Clearly, one can also construct the transformation functions $T_g(\cdot)$, $T_h(\cdot)$ such that a violation of a constraint is weighted as a function of time, of any independent variable, or of any dependent variable. If one wants for example minimize energy consumption, then it would be reasonable to impose punishment for not satisfying thermal comfort only when the building is occupied, i.e., during daytime on weekdays.

The advantage of this approach is that it allows free definition of coupled nonlinear constraints that have to be satisfied. In contrast to the method described in Section 7.1, (7.14) also allows constraining a dependent variable at any time during the simulation. However, selecting appropriate transformation functions $T_g(\cdot)$ and $T_h(\cdot)$ and selecting w_g and w_h is not trivial. The rate of convergence is strongly affected by this selection. In particular, problems might occur if $g(\cdot)$ and its derivatives are not continuous at zero.

To implement the above dependent variable constraints, we usually use barrier or penalty functions. Note, however, that even if the barrier and penalty functions are described in terms of restricting the dependent variables, it is clear that the same technique can also be applied to independent variables.

In the explanation below, if no ambiguity arises, we will often write w instead of w_g and w_h to facilitate the notation.

7.2.1 Barrier Functions

Barrier functions impose a strong punishment if the dependent variable gets close to the boundary of the feasible region. The closer the variable is to the boundary, the higher the value of the barrier function becomes. The general form of a barrier function is $T_g^i\big(g^i(x)\big) \geq 0$, with $T_g^i\big(g^i(x)\big) \to \infty$ as $g^i(x) \to 0$ for all i.
A possible form of a barrier function is

$$T_g^i\big(g^i(x)\big) = \frac{1}{\big(g^i(x)\big)^{2\,r}}, \qquad r \geq 1. \tag{7.15}$$

One drawback of the barrier function is that the boundary of the feasible set and its immediate neighborhood can, assuming a perfect numerical implementation, never be reached. Moreover, if the variation of the independent variable between two iterations is too big, we can actually cross the boundary. A robust implementation of a barrier function catches such numerical problems

and sets the independent variable back to a feasible value. However, this produces oscillations in the objective function that badly affect the convergence rate and can even cause divergence.

Another drawback of the barrier function is that it does not allow formulating equality constraints.

Different approaches are possible for the weighting factors, w_g:

a) Variation of the Weighting Factors

As already mentioned, one drawback of barrier functions is a punishment is imposed when we get close to the boundary of the feasible domain even if we are still in the feasible set. The smaller the weighting factors, the smaller is the punishment added to the objective function for a given set of parameters. In the extreme case we can get infinitesimally close to the boundary, as w tends to zero. However, it is obvious that numerical difficulties will occur for very small weighting factors. To overcome this problem, we can start with a moderately large w_0 and let w_k tend to zero during the optimization process, i.e.,

$$w_0^i > \ldots > w_k^i > w_{k+1}^i > \ldots > 0, \tag{7.16}$$

where k denotes the iteration number and i the element of w.

However, we are left with choosing the proper value of w_0 and the decrements that reduce the sequence w_k to zero as $k \to \infty$. For instance, if w_k decreases too fast the problem might become ill conditioned, which causes numerical difficulties. On the other hand, w_k's that are too big cause too high a punishment, so we cannot get close enough to the boundary of the feasible set.

A more effective technique is to extrapolate the parameter values and the objective function value based on optimal values for some given constants, w, as shown in the next section.

b) Extrapolation

Extrapolating the optimal free parameters and the corresponding value of the objective function, based on some points for given, fixed w_k's, is a simple approach for approximating the optimal parameter set x and $f(x)$ for $w_k \to 0$. Using this technique instead of successive reduction of the weighting factors reduces calculation time.

The idea is to optimize the objective function for a given number (for example, 3) of weighting factors. Once the optimization is done, a function, say $p^i(w)$, $i \in \{1, \ldots, n\}$ is put through the known points of each parameter x^i. The optimal parameter set is then obtained by evaluating $p(w = 0)$. To get the objective function value for $x^* = p(0)$, the same technique can be used for the objective function value, or the simulation can be run again with $x = x^*$ and barrier functions removed.

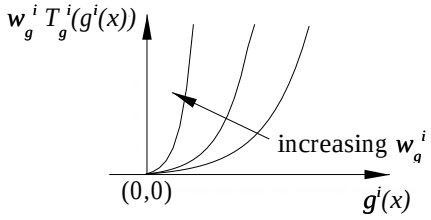


Fig. 7.2: Penalty function for inequality constraint

7.2.2 Penalty Functions

In contrast to barrier functions, penalty functions allow crossing the boundary of the feasible set. As soon as the boundary is crossed, a positive penalty term is added to the objective function. Penalty functions allow equality constraints to be imposed (which is not possible with barrier functions). Fig. 7.2 shows an example of a penalty function with different weighting factors w_k, where k stands for the iteration number.

The general form of a penalty function is given for each component $T_g^i\big(g^i(x)\big)$ of $T_g\big(g(x)\big)$ and $T_h^i\big(h^i(x)\big)$ of $T_h\big(h(x)\big)$ by

$$T_g^i\big(g^i(x)\big) \begin{cases} > 0, & \text{if } g(x) > 0, \\ = 0, & \text{otherwise,} \end{cases} \tag{7.17}$$

$$T_h^i\big(h^i(x)\big) \begin{cases} > 0, & \text{if } h(x) \neq 0, \\ = 0, & \text{otherwise,} \end{cases} \tag{7.18}$$

where each element $T_g^i(\cdot)$ and $T_h^i(\cdot)$ is continuous and monotonically increasing as one gets further inside the unfeasible set.

Possible penalty functions for inequality constraints are

$$T_g^i\big(g^i(x)\big) = \begin{cases} \exp\big(g^i(x)\big) - 1, & \text{if } g(x) > 0, \\ 0, & \text{otherwise,} \end{cases} \tag{7.19}$$

or

$$T_g^i\big(g^i(x)\big) = \begin{cases} \big(g^i(x)\big)^{2\,r}, \quad r \geq 1, & \text{if } g(x) > 0, \\ 0, & \text{otherwise,} \end{cases} \tag{7.20}$$

and, for equality constraints,

$$T_h^i\big(h^i(x)\big) = \exp\Big(\big(h^i(x)\big)^2\Big) - 1, \tag{7.21}$$

or

$$T_h^i\big(h^i(x)\big) = \big(h^i(x)\big)^2, \quad r \geq 1. \tag{7.22}$$

The disadvantage of this approach is that the derivative of $T_g^i(\cdot)$ is not continuous at the boundary of the feasible set. Depending on the optimization algorithm, this can produce oscillations during the optimization process, or even leading to non-convergence.

As in the barrier method, selecting the weighting factors w_g and w_h is not trivial. Too small a value for w produces too big a trade-off. Hence, the boundary of the feasible set can be exceeded by an unacceptable amount. On the other hand, too large a value of w leads to ill conditioning of the optimization problem and so can cause numerical problems.

Unlike the case with barrier functions, the weighting factors must not tend to zero, otherwise the penalty term will vanish. The weighting factors have to satisfy the inequality

$$0 < w_0 < \ldots < w_k < w_{k+1}. \tag{7.23}$$

7.2.3 Slack Variables

The method of slack variables ensures that the transition of an inequality constraint and its derivatives is smooth at the boundary of the feasible set. This is accomplished by introducing, for each inequality constraint, $g^i(x)$ with $i \in \{1, \ldots, k\}$, a new free parameter, s^i with $i \in \{1, \ldots, k\}$. The general form of the transformation function is then

$$T_g^i\big(g^i(x)\big) = \big(g^i(x) + (s^i)^2\big)^{2\,r}, \quad r \geq 1, \tag{7.24}$$

where the optimization is now done with respect to the new set of free parameters, $\hat{x} \triangleq (x,\, s) \in \mathbb{R}^{n+k}$.

The slack variables s^i can have any value. They are treated in the optimization as any other independent variable x^i. If an inequality constraint is not violated, that is $g^i(x) \leq 0$, then s^i will be set to $-(g^i(x))^{0.5}$, and hence $T_g^i\big(g^i(x)\big)$ vanishes. On the other hand, if the i-th inequality constraint is violated, then s^i will tend to zero and $T_g^i\big(g^i(x)\big)$, multiplied by w_g^i, increases the value of the objective function.

If we also have to deal with equality constraints, they can simply be introduced in the form of penalty functions.

As with penalty functions, the solution of the modified objective function converges to the solution of the original constrained problem, $f(x)$, as w tends to infinity.

Since each additional slack variable causes the optimization to be done in a higher dimensional space, the number of slack variables should be kept as small as possible.

7.2.4 Implementation of Barrier Functions, Penalty Functions, and Slack Variables

Since, according to (7.16) and (7.23), the weighting factor, w, depends on the number of the optimization run, we need to know the iteration number in order to increase or decrease w. However, the weighting factors cannot simply be changed in every simulation run. Since for special purposes, like approximating the gradient or performing a line search, the *same* objective function has to be evaluated several times, we have to fix w during certain simulation runs.

One way to do this is to divide the simulation calls into *main* and *sub* iterations. However, this may be confusing for some algorithms since it is not always clear how to distinguish between a simple trial of a parameter set and an effective move towards the minimum.

To overcome these difficulties a method is implemented into the class `Optimizer` that increments a counter in each call. If the keyword `WriteStepNumber` in the optimization command file is set to `true`, the method calls the simulation to evaluate the objective function for the new value of this counter. If `WriteStepNumber` is `false`, no new function evaluation is performed by this method since the objective function does not depend on this counter.

To vary the weighting factors, one can simply read this counter in the simulation program and assign new values to the weighting factors.

7.3 Summary

We have described different methods for taking constraints into account. In the simplest case of box constraints, a simple transformation of the constrained parameters is performed. The optimization is then carried out in this new, unconstrained space.

Coupled linear constraints are rare in building simulation. They are most efficiently handled by specially constructed optimization algorithm.

To take non-linear (coupled) constraints on independent and dependent variables into account, one can use barrier or penalty functions, or for inequality constraints, also introduce slack variables. Penalty methods approximate a constrained problem by an unconstrained problem by assigning a high cost to the objective function if one gets out of the feasible set. In contrast to the penalty methods, the barrier methods do not allow leaving the feasible set. These methods assign a high cost to the objective function if one gets close to the boundary. Barrier methods cannot be used for equality constraints since with barrier methods, the restricted variables have to be feasible all the times.

One problem with the penalty function for inequality constraints is that its derivative is not continuous at the boundary of the feasible set. To overcome this problem one can reformulate the inequality transformation by introducing slack variables. However, the smooth transition is offset by the higher number of free parameters that have to be optimized.

The solution of the modified unconstrained problem with barrier functions, penalty functions and slack variables converges to the solution of the original constrained problem as the weighting factors w tend to zero (for barrier functions) or to infinity (for penalty functions and slack variables). However, as w tends to zero (for barrier functions) or infinity (for penalty functions and slack variables), the objective function becomes ill conditioned for all three methods.

Input Files

initialization:	Specification of file location (input files, output files, log file, etc.)
command:	Specification of parameter names, initial values, bounds, optimization algorithm, etc.
configuration:	Configuration of simulation program (error indicators, start command, etc.)
simulation input template:	Templates of simulation input files

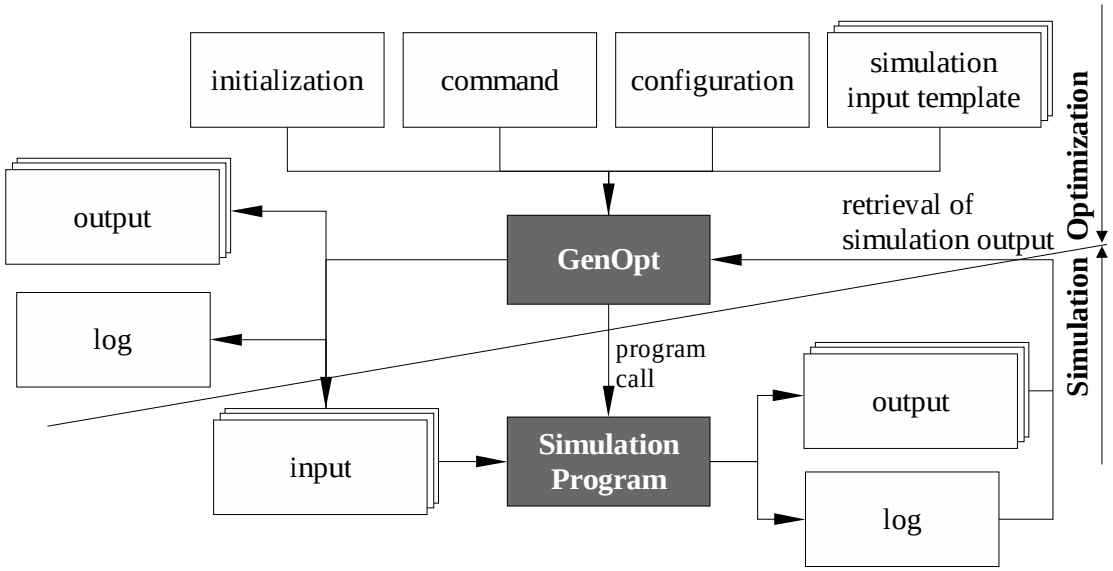


Fig. 8.1: Interface between GenOpt and the simulation program that calculates the objective function

8 Program

GenOpt 1.1.2 is written entirely in Java 2 v1.4.0 to ensure platform independence. GenOpt is divided into two main parts. One part does all the data handling, storing the results and communicating with the simulation program. The other part is the actual optimization portion. It contains the optimization algorithms, auxiliary classes that are used for performing the optimization calculations like testing the optimality condition, performing a line search, etc.

Since there is a variety of simulation programs and optimization algorithms, GenOpt is kept entirely open on both sides. This allows easy implementation of new optimization algorithms as well as using any software to perform the simulation.

8.1 Interface to the Simulation Program

Text files are used to exchange data with the simulation program and to specify how the simulation program is to be started. This makes it possible to couple any simulation program to GenOpt without requiring code adaptation

on either the GenOpt side or the simulation program side.

The simulation program must satisfy the following requirements:

1. It must be capable of reading input from one or more text files, writing the value of the objective function to a text file, and writing error messages to a text file.
2. It must be able to be launched automatically by a command and must terminate automatically. This means that the user does not have to open the input file manually and shut down the simulation program once the simulation is finished.

The simulation program may be a commercially available program or one written by the user.

8.1.1 Post-Processing of the Objective Function Value

In some optimization problems, the objective function value may depend on different simulation results. For instance, you may want to minimize the (weighted) sum of annual heating and cooling energy consumption, which we will call *total energy*. Some simulation programs, such as SPARK, TRNSYS, etc. allow computing the total energy directly. Other simulation programs, such as EnergyPlus, cannot add different output variables. In this case, you can only write the heating and cooling energy consumption separately to the output file. In order to be able to optimize the total energy, the simulation output must be post-processed.

If you want to post-process the objective function value in GenOpt, you could proceed as follows:
Suppose the objective function delimiter (see Section 10.1.1) for the heating and cooling energy are, respectively, `Eheat=` and `Ecool=`. Then, you can specify in the optimization initialization file (see Section 10.1.1) the section

```
ObjectiveFunctionLocation{
   Delimiter1 = "Eheat=";  Name1 = "E_tot";
   Delimiter2 = "Eheat=";  Name2 = "E_heat";
   Delimiter3 = "Ecool=";  Name3 = "E_cool";
}
```

The file `Optimizer.java` has a method `postProcessObjectiveFunction(int, double[])` that is called after the objective function value has been read from the simulation output files. The arguments of this method are the iteration number and an array that contains the objective function values. In our example, the array has three elements. The 0th and 1st element contains the value of the heating energy, and the 2nd element the value of the cooling energy. To minimize the total energy, you can overwrite this method as follows:

```
private void postProcessObjectiveFunction(int iterationNumber,
                                          double[] f){
  f[0] = f[1] + f[2];
  if (iterationNumber == 1)
     setInfo("Post process objective function value.");
  return;
}
```

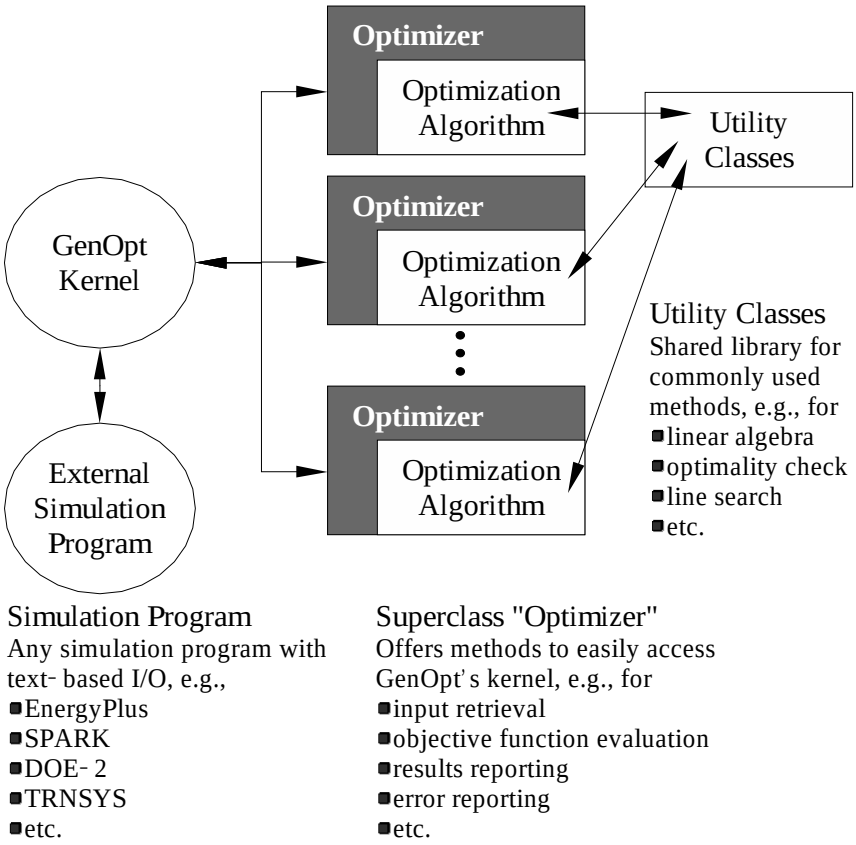


Fig. 8.2: Implementation of optimization algorithms into GenOpt

Hence, the 0th element of the array `f` contains after the method call the total energy. Since GenOpt's minimization algorithms minimize only the 0th element of `f` (the other elements are only being used for report), we now optimize the total energy. GenOpt will also write an information to the log file and the user interface to remind that the objective function value is being post processed.

After making the above changes, the file `genopt/algorithm/Optimizer.java` needs to be compiled. To compile this file, you need to have a Java compiler (such as the one from Sun Microsystems). To compile it, open a console (or DOS window), change to the directory `genopt/algorithm` and type

```
javac Optimizer.java
```

This will generate the file `Optimizer.class`. If the compilation fails, you most likely did not specify the `CLASSPATH` variable as described in Chapter 9.

8.2 Interface to the Optimization Algorithm

The large variety of optimization algorithms leads to an open interface on the algorithm side. To do the optimization, one can couple any Java class designed according to the guidelines of Section 8.4. Thus, users can implement

their own algorithms and add them to the library of available optimization algorithms without having to adapt and recompile GenOpt. This means that GenOpt can not only be used to do optimization with built-in algorithms, but it can also be used as a framework for developing, testing and comparing optimization algorithms.

The class `Optimizer` is the superclass of each optimization algorithm. It offers all the functions required for easy retrieval of parameters that specify the optimization settings, performing the evaluation of the objective function and reporting intermediate results. For a listing of its methods, see `http://simulationresearch.lbl.gov` or the Javadoc code documentation that comes with GenOpt's installation.

8.3 Package `genopt.algorithm`

The Java package `genopt.algorithm` consists of all the classes that are directly used to define the optimization algorithm. It is structured as follows:

`genopt.algorithm` This package contains all optimization algorithms. The superclass `Optimizer`, which must be inherited by each optimization algorithm, is part of this package.

`genopt.algorithm.optimality` This package contains classes that can be used to check whether a parameter set is at a minimum point or not.

`genopt.algorithm.gradient` This package contains classes that can be used for approximating the gradient of the objective function. None of these classes have been implemented yet.

`genopt.algorithm.linesearch` This package contains classes for doing a line search along a given direction.

`genopt.algorithm.util.math` This package contains classes for mathematical operations.

The Javadoc source code documentation comes with GenOpt's installation. It can also be seen on `http://simulationresearch.lbl.gov`.

8.4 Implementing a New Optimization Algorithm

The Java class that represents the optimization algorithm must have the form shown in Fig. 8.3 and must use the methods of its superclass `Optimizer` to evaluate the objective function and report the optimization steps. The methods of the `Optimizer` class are listed in `http://simulationresearch.lbl.gov`.

To implement and use your own algorithm, follow these steps:

1. Place the bite-code (*ClassName*`.class`) in the directory `genopt/algorithm` (on Linux or Unix) or `genopt\algorithm` (on Windows).
2. Set the value of the keyword `Main` in the `Algorithm` section of the optimization command file to the name of the optimization class (without file extension).

```
package genopt.algorithm;

import genopt.GenOpt;
import genopt.lang.OptimizerException;
import genopt.io.InputFormatException;

public class ClassName extends Optimizer{

   public ClassName (GenOpt genOptData)
      throws OptimizerException, IOException,
      InputFormatException
   {
      // set the mode to specify whether the
      // default transformations for the box
      // constraints should be used or not
      int constraintMode = xxxx;
      super(genOptData, constraintMode);

      // remaining code of the constructor
   }

   public int run() throws OptimizerExcep-
tion, IOException
   {
      // the code of the optimization algorithm
   }

   // add any further methods and data members
   // you need in your algorithm
}
```

Fig. 8.3: Code snippet that specifies how to implement an optimization algorithm

3. Add any further keywords you require to the `Algorithm` section. The keywords must be located *after* the entry `Main` of the optimization command file. The keywords must be in the same sequence as they are called in the optimization code.
4. Do not forget to call the method `Optimizer.report()` after evaluating the objective function. Otherwise, the result will not be reported.
5. In order to allow the implementation of a variation of the weighting factors used for penalty or barrier functions and slack variables, the method `Optimizer.increaseStepNumber()` must be called from the optimization code. You must call this method every time a whole iteration step is completed but *not* during an optimization stage in which the definition of the objective function must stay exactly the same (such as during a line search or a gradient approximation).

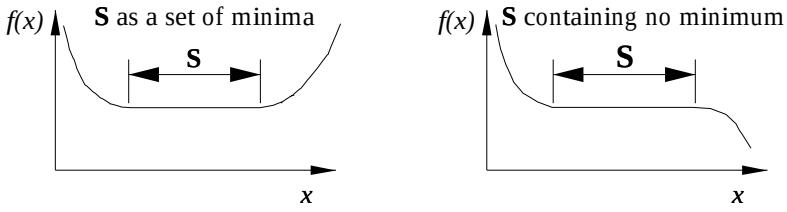


Fig. 8.4: Examples of a null space with and without minima

8.5 Handling of Null Space of the Objective Function

Let the $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a twice continuously differentiable objective function. A null space of $f(\cdot)$ is an open subset, $\mathbf{S} \subset \mathbb{R}^n$, of the parameter space given by

$$\mathbf{S} \triangleq \{x \mid \nabla f(x) = 0 \wedge \det H(x) = 0\} \tag{8.1}$$

where H denotes the Hessian matrix (6.4), as defined on page 13. The set $\mathbf{S}$ might be the set of (local) minima points of the function. Two examples of the set $\mathbf{S}$ are given in Fig. 8.4 for the one-dimensional case $f : \mathbb{R} \rightarrow \mathbb{R}$.

In the null space $\mathbf{S}$, no information about the descent direction of the function can be obtained. If the algorithm gets in the set $\mathbf{S}$, then no guarantee can be given that the minimum of the function can be reached. This leads to either no convergence or in the best case to a slower rate of convergence.

Null spaces often occur if the objective function is written with only a few significant digits to the output file. Truncating significant digits causes problems particularly towards the end of the optimization when the change of the objective function is small. For example, suppose that an objective function value of $f(x) = 1/3$ is written as `0.33`. Assume that the change of the objective function in the current stage of the optimization affects only the third digit beyond the decimal point. Then no useful information about the function can be obtained at this stage.

To detect such cases, the optimization algorithm can force GenOpt to check if objective function values are equal. To do the check, all *checked* values are stored internally. The check is then performed among those values *only*. That means, that only if the algorithm asks for a check after a function evaluation, this particular function value will be part of the set for which later checks will be performed. The separation between values that are checked and are not is done because some function values might be obtained repeatedly, i.e., in stages where the scaling of the problem has to be detected.

If the same function value is obtained more than a specified number of times, then GenOpt terminates with an error message. The maximum number of equal function values can be specified by the setting of `MaxEqualResults` in the command file (see page 65). For informational purposes, all objective

function values that have been checked previously are reported to the user interface and to the log file.

9 Installing and Running GenOpt

9.1 Installing GenOpt

To use GenOpt, a Java interpreter for Java 2 v1.4.0 or higher must be installed, such as Sun's Java Runtime Environment (JRE). If you want to add your own optimization algorithm you must also have a Java compiler. Sun's Java Development Kit (JDK) contains a compiler and the runtime environment. GenOpt has been tested with Sun's Java 2 v1.4.0 .

You can download JRE and JDK from `http://java.sun.com/products/`. JRE is freeware. JRE and JDK also run on the Windows operating system.

Once you have installed Java on your system, you need to download the file `go_prg.linux` (self extracting file, for Linux only) or `go_prg.zip` from `http://SimulationResearch.lbl.gov` onto your system and extract it. To extract the file `go_prg.linux` on Linux, type

```
chmod +x go_prg.linux
./go_prg.linux
```

To extract `go_prg.zip` on Unix, type

```
unzip go_prg.zip go_prg
```

On Windows, you can use the software WinZip to extract the file `go_prg.zip`.

9.2 System Configuration for JDK Installation

In the instructions below, "`.`" stands for the current directory and the directory `go_prg` contains the directory `genopt` (i.e., `go_prg/genopt`) where the Java class files of GenOpt are stored.

9.2.1 Linux/Unix

The installation is explained for the bash-shell and the C-shell. In both cases, it is assumed that `/usr/local/jdk/bin` is the directory that contains the Java binaries. If you use the bash-shell you have to add the following lines to the `~/.bashrc` file:

```
PATH="$PATH:/usr/local/jdk/bin"
CLASSPATH="$CLASSPATH:.:$HOME/go_prg"
export PATH CLASSPATH
```

If you use the C-shell, you have to add the following lines to the `~/.cshrc` file:

```
set PATH=PATH:/usr/local/jdk/bin
setenv CLASSPATH CLASSPATH:.:$HOME/go_prg
```

9.2.2 Microsoft Windows

The `PATH` variable of the system must contain the directory where the Java Virtual Machine is located. For example,

```
SET PATH=%PATH%;C:\prog\jdk\bin
```

There must also be a CLASSPATH variable that points to the Java classes. This variable has the form

```
SET CLASSPATH=%CLASSPATH%;.;C:\prog\go_prg
```

In Windows 2000 both variables can be specified under "Start → Settings → Control Panel → System → Advanced Environment Variables". In Windows 95 and 98 they can be specified in the `autoexec.bat` file.

9.3 Starting an Optimization with JDK Installation

GenOpt can be run either with a graphical user interface (GUI), or as a console application. The GUI features an online chart showing the optimization progress of the objective function and of certain parameters. Any values can be added or removed from the chart during runtime. The console application allows running GenOpt with a batch job for several sequential optimizations or starting GenOpt over a remote connection, e.g., using `telnet`. You can start the GUI version of GenOpt with the command

```
java genopt.WinGenOpt
```

and the console version with

```
java genopt.GenOpt [Optimization Initialization File]
```

In these commands, `java` is the name of the Java virtual machine (that interprets the byte code), and `genopt.WinGenOpt` (or `genopt.GenOpt`) is the full name of the main class. The brackets indicate that the last parameter is optional. The optimization initialization file can also be specified after launching GenOpt.

9.4 System Configuration for JRE Installation

First, make sure that the path variable points to the path where the `jre` binary (i.e., `jre.exe`) is located. If this is not the case, you have to set the path variable as described in Section 9.2.

It is recommended that you set an environment variable, similar to the `CLASSPATH` variable, to the `genopt` directory. This can be done in the same way as described for the `CLASSPATH` variable for the JDK installation. Note that on Windows platforms, JRE will ignore the `CLASSPATH` environment variable. For both Windows and Solaris platforms, the `-cp` option is recommended to specify an application's class path.

9.5 Starting an Optimization with JRE Installation

The GUI version can be launched with the command

```
jre -cp %CLASSPATH% genopt.WinGenOpt
```

and the console version with

```
jre -cp %CLASSPATH% genopt.GenOpt [OptInitializationFile]
```

where `CLASSPATH` is the name of the environment variable that points to the `genopt` directory.

10 Setting Up an Optimization Problem

The first step in specifying an optimization problem is to define the objective function to be minimized within an external program that satisfies the requirements listed on page 49. If you have to maximize your objective function, you can change the sign of the objective function to turn the maximization problem into a minimization problem.

During the optimization the simulation that calculates the objective function has to be executed several times, and in many cases, hundreds or even thousands of times. Therefore, it is worthwhile to simplify your problem as much as possible. In particular, if you want to minimize annual energy consumption, you might calculate representative days to get a first guess of the optimum point if the annual simulation takes too long. You can then use the guess as a starting point for an optimization over the whole year.

Besides defining the objective function, you have to specify the range allowed for each of the free parameters. If the ranges are restricted, then you can use the default scheme for box constraints or modify your problem with penalty terms, barrier terms or slack variables as described in Chapter 7.

Once you have set up your objective function, you have to write its value into the simulation output file. *It is important that the objective function value is written to the output file without truncating significant digits* (see page 53). For example, if you calculate the objective function in Fortran double precision, it is recommended that you use `E24.16` format.

To indicate which value in simulation output file is the value of the objective function, you can specify it with a leading string (see page 62).

Once you have finished specifying your simulation, you have to specify the files described in the following section.

10.1 File Specification

This section defines the file syntax for GenOpt. Please see also the files in the directory `example` of the GenOpt installation.

The syntax of the GenOpt files is structured into sections of parameters that belong to the same object. The sections have the form

```
ObjectKeyWord { Object }
```

where `Object` can either be another `ObjectKeyWord` (nesting of objects) or an assignment of the form

```
Parameter = Value ;
```

Some variables allow being referenced. References have to be written in an object-like manner, i.e., in the form

```
Parameter = ObjectKeyWord1.ObjectKeyWord2.Value ;
```

where `ObjectKeyWord1` refers to the root of the object hierarchy as specified in the corresponding file.

To clarify the different possible values for keywords, the following notation is introduced:

1. Text that is part of the file is shown in `fixed width fonts`.
2. `|` stands for possible entries. Only one of the entries that are separated by `|` is allowed.
3. `[ ]` indicates optional values.
4. The file syntax follows the Java convention. This means,
   (a) `//` indicates a comment on a single line,
   (b) `/*` and `*/` enclose a comment,
   (c) the equal sign, `=`, assigns values,
   (d) a statement has to be terminated by a semi-colon, `;`,
   (e) curly braces, `{ }`, enclose a whole section of statements, and
   (f) the syntax is case sensitive.

The following basic types are used:

`String`	any sequence of characters. If the sequence contains a blank character, it has to be enclosed in apostrophes ("). If there are apostrophes within quoted text, they must be specified by a leading backslash (i.e., \"). Similarly, backslash within quoted text must be specified by a leading backslash (e.g., `c:\prog\genopt`).
`StringReference`	Any name of a variable that appears in the same section.
`Integer`	Any integer value.
`Double`	Any double value (including integer)
`Boolean`	Either `true` or `false`

10.1.1 Initialization File

The initialization file specifies where the *files* of the current optimization problems are located, which simulation files must be saved, and what additional strings have to be passed to the simulation call. It also specifies which simulation program is used by having a variable that points to the simulation configuration file, which in turn specifies the command that launches the simulation program.

The sections must be specified in the order shown in the following example. The order of the keywords in each section is arbitrary.

The initialization file syntax looks like:

```
Simulation {
  Files {
    Template {
      File1 = String | StringReference;
      Path1 = String | StringReference;
     [File2 = String | StringReference;
      Path2 = String | StringReference;
     [ ... ] ]
    }
    Input { // the number of input file must be equal to
            // the number of template files
      File1     = String | StringReference;
      Path1     = String | StringReference;
     [SavePath1 = String | StringReference;]
     [File2     = String | StringReference;
      Path2     = String | StringReference;
     [SavePath2 = String | StringReference;]
     [ ... ] ]
    }
    Log {
      File1     = String | StringReference;
      Path1     = String | StringReference;
     [SavePath1 = String | StringReference;]
     [File2     = String | StringReference;
      Path2     = String | StringReference;
     [SavePath2 = String | StringReference;]
     [ ... ] ]
    }
    Output {
      File1     = String | StringReference;
      Path1     = String | StringReference;
     [SavePath1 = String | StringReference;]
     [File2     = String | StringReference;
      Path2     = String | StringReference;
     [SavePath2 = String | StringReference;]
     [ ... ] ]
    }
    Configuration {
      File1 = String | StringReference;
      Path1 = String | StringReference;
    }
  } // end of section Simulation.Files
  [CallParameter {
    [Prefix = String | StringReference;]
    [Suffix = String | StringReference;]
  }]
  [ObjectiveFunctionLocation {
    Delimiter1  = String | StringReference;
    Name1       = String;
    [Delimiter2 = String | StringReference;
     Name2      = String;
    [ ... ] ]
```

```
  }]
} // end of section Simulation
Optimization {
  Files {
    Command {
      File1 = String | StringReference;
      Path1 = String | StringReference;
    }
  }
} // end of section Optimization
```

The sections have the following meaning:

`Simulation.Files.Template` The set of free parameters that is optimized by the optimization program will be written in the simulation input files. To do so, GenOpt reads the simulation input *template* files, replaces each occurrence of `%variableName%` by the numerical value of the corresponding variable, and the resulting file contents are written as the simulation input files. The string `%variableName%` refers to the name of the variable as specified by the entry `Name` in the optimization command file on page 65.

If the free parameters have to be written to several simulation input files, this can be specified by adding as many `Filei` and `Pathi` assignments as necessary – where `i` stands for a one-based counter of the file and path – to the `Template` section. Note that there must obviously be the same number of files and paths in the `Input` section that follows this section.

In case of multiple simulation input template files, each file will be written to the simulation input file whose keyword ends with the same number.

The following rules are imposed:

1. Each variable name specified in the optimization command file *must* occur in at least one simulation input template file.
2. Multiple occurrences of the same variable name are allowed in the same file or also in different files.
3. If the value `WriteStepNumber` in the section `OptimizationSettings` of the optimization command file is set to `true`, then the string `%stepNumber%` must occur in at least one simulation input template file. Rule 1 and 2 apply also to `%stepNumber%`. If `WriteStepNumber` is set to `false`, then `%stepNumber%` can occur, but it will not be replaced.

`Simulation.Files.Input` The simulation input file is automatically generated by GenOpt based on the current parameter set and the corresponding simulation input *template* file, as explained in the previous paragraph. Obviously, the number of simulation input files must be equal to the number of simulation input template files.

The section `Input` has an optional key word, called `SavePath`. If `SavePath` is specified, then the corresponding input file will after each simulation

be copied into the directory specified by `SavePath`. The copied file will have the same name, but with the simulation number added as prefix.

`Simulation.Files.Log` The simulation log file is scanned for error messages. The optimization terminates if any of the strings specified by the variable `ErrorMessage` in the `SimulationError` section of the GenOpt configuration file is found. At least one log file must be specified.

The section `Log` also has the optional key word `SavePath`. It has the same functionality as in the previous section.

`Simulation.Files.Output` The value of the objective function is read from this file. The value that is written after the *last* occurrence of the string specified by `Delimiter1` in the section `ObjectiveFunctionLocation` is regarded as the value of the objective function. The number of objective function values is arbitrary (at least one must be specified). The currently implemented optimization algorithms minimize the first objective function value. The other values are only reported to the output files and the online chart.

GenOpt searches for the objective function value as follows:

1. After the first simulation, GenOpt searches for the first objective function value in the first output file. The number that occurs after the *last* occurrence of the string specified by the variable `Delimiter1` in the section `ObjectiveFunctionLocation` is regarded as the value of the objective function. Only if the first output file does not contain the first objective function value will GenOpt proceed with reading the second output file (if present) and so on until the last output file is read. If GenOpt cannot find the objective function value in any of the output files, it will terminate with an error. The same procedure is repeated with the second objective function value (if present) until all objective function values have been found.
2. In the following iterations, GenOpt will only read the file(s) where it found the objective function value(s) after the first simulation. The other output files are not read.

This section also contains the optional keyword `SavePath`. If this keyword is specified, then GenOpt copies the output file, regardless of whether it contains the objective function value or not. This is particularly useful for doing parametric runs, e.g., with the algorithm `EquMesh`, page 39.

`Simulation.Files.Configuration` The entries in this section specify the simulation configuration file, which contains information that is related to the simulation program only, but not related to the optimization problem. The simulation configuration file is explained below.

`Simulation.CallParameter` Here, a prefix and suffix for the command that starts the simulation program can be added. With these entries, any additional information, such as the name of weather files, can be passed to the simulation program. To do so, one has to refer to either of these entries in the argument of the keyword `Command`, page 64.

`Simulation.ObjectiveFunctionLocation` This section specifies where the values of the objective function can be found in the simulation output file. GenOpt reads the value after the *last* occurrence of `Delimiteri` (where `i` stands for `1`, `2`, `3`, `...`) as the objective function value. The value of `Namei` has no other functionality than labeling the results.

For convenience, the section `ObjectiveFunctionLocation` can optionally be specified in the initialization file, but its specification is required in the configuration file. If this section is specified in both files, then the specification in the initialization file will be used.

Specifying the section `ObjectiveFunctionLocation` in the initialization file is of interest if a simulation program is used for different problems that require different values of this section. Then, the same (program specific) configuration file can be used for all runs and the different settings can be specified in the (project dependent) initialization file rather than in the configuration file.

`Optimization.Files.Command` This section specifies where the optimization command file is located. This file contains all mathematical information of the optimization. It is described further on page 65.

10.1.2 Configuration File

The configuration file contains information related only to the simulation program used and not to the optimization problem. Hence, it has to be written only once for each simulation program and operating system. It is advisable to put this file in the directory `go_prg/genopt/cfg` so that it can be used for different optimization projects. Some configuration files are provided with the GenOpt installation.

The syntax is specified by

```
// Error messages of the simulation program
SimulationError{
   ErrorMessage = String;
  [ErrorMessage = String;
  [ ... ] ]
}

// Number format for writing simulation input files and result files
IO{
   NumberFormat = Float | Double;
}

// Specifying what command launches the simulation
SimulationStart{
   Command = String;
   WriteInputFileExtension = Boolean;
}

// Specifying the location of the
// objective function value in the simulation output file
ObjectiveFunctionLocation{
    Delimiter1 = String | StringReference;
    Name1      = String;
    [Delimiter2 = String | StringReference;
     Name2      = String;
     [...]]
}
```

The entries have following meaning:

`SimulationError` The error messages that might be written by the simulation program must be assigned to the keyword `ErrorMessage` so that GenOpt can check whether the simulation has completed successfully. At least one entry for `ErrorMessage` must be given.

`IO` The keyword `NumberFormat` specifies in what format the design parameters will be written to the simulation input file. The setting `Double` is recommended, unless the simulation program cannot read this number format.

`SimulationStart` The keyword `Command` specifies what string must be used to start the simulation program and wait for its termination. The value of the variable `Command` is treated in a special way: Any value of the optimization initialization file can be automatically copied into the value of `Command`. To do so, you have to surround the reference to the corresponding keyword with percent signs. A reference to the keyword `Prefix` of the initialization file would, for example, look like

```
%Simulation.CallParameter.Prefix%
```

By setting `WriteInputFileExtension` to `false`, the value of the keyword `Simulation.Input.Filei` (where `i` stands for `1`, `2`, `3`) is copied into `Command`, and the file extension is removed.

`ObjectiveFunctionLocation` Note that this section can also be specified in the initialization file, where its values override the settings of the configuration file. This section has been described on page 63.

10.1.3 Command File

The command file specifies optimization-related settings such as the free parameters, stopping criteria and optimization algorithm. The sequence of the entries in all sections of the command file is arbitrary.

The structure is:

```
// Settings of the free parameters
Vary{
   // Parameter entry
   Parameter{
      Name = String;
      [ Min  = Double | SMALL; ]
      Ini  = Double;
      [ Max  = Double | BIG; ]
      Step = Double;
   }
   ...
}

// General settings for the optimization process
OptimizationSettings{
   MaxIte          = Integer;
   WriteStepNumber = Boolean;
   [ MaxEqualResults = Integer; ]
}

// Specification of the optimization algorithm
Algorithm{
   Main = String;
   ... // any other entries that are required
       // by the chosen optimization algorithm
}
```

The different sections are:

`Vary` This section contains the set of free parameters.

`Parameter` Here the properties of the parameter are listed. `Name` specifies the name of the variable. The simulation input template files will be scanned for this string – surrounded by percent signs – and each occurrence will be replaced with its numerical value before writing the simulation input files. The keywords `Min` and `Max` determine the lower and upper bound, respectively, for each parameter. If the keywords are omitted or set to `SMALL` and `BIG`, the parameter is regarded as unconstrained. `Ini` specifies the initial value and `Step` the step size. Even if not all optimization algorithms use a step size, it should be set to an appropriate value. An algorithm may use this value as an indicator about the scaling of the variable, i.e., for making an initial estimate of the length over which a line search should be performed.

`OptimizationSettings` This section specifies general settings of the optimization. `MaxIte` is the maximum number of iterations. If more than `MaxIte` main iterations are performed, GenOpt terminates with an error message. `WriteStepNumber` indicates whether the current step of the optimization has to be written in the simulation input file. The step number can then be used to calculate a barrier or penalty function in the simulation program (see Section 7.2 on page 42). The optional parameter `MaxEqualResults` specifies how many times the objective function can equal a value that has already been previously obtained before GenOpt terminates. This setting is used to terminate GenOpt when the objective function has a null space (see page 53). The default value of `MaxEqualResults` is 5.

`Algorithm` The setting of `Main` specifies which algorithm is invoked for performing the optimization. Its value has to be equal to the class name that contains the algorithm. Note that additional parameters might be required depending on the algorithm used (see Section 6.3 on page 19 for the implemented algorithms).

10.1.4 Log File

The GenOpt log file contains general information about the optimization process. It also contains warnings and errors that occur during the optimization.

GenOpt writes the log file to the directory that contains the initialization file. The name of the log file is `GenOpt.log`.

10.1.5 Output File

GenOpt writes two output files (in addition to `GenOpt.log`). Both of the files `OutputListingMain.txt` and `OutputListingAll.txt` list the optimization steps; the first contains only the main iteration steps and the second all iteration steps. The files are written to the directory where the optimization command file is located (specified by the variable `Optimization.Files.Command.Path1` in the optimization initialization file).

Both output files are generated automatically by GenOpt. Each time the method `genopt.algorithm.Optimizer.report()` is called from the optimization algorithm, the current trial is reported in either one of the files.

11 Further Development

The following enhancements to GenOpt are planned:

1. Extending the library of optimization algorithms so that, depending on the problem structure, additional algorithms can be selected.
2. Development of an efficient optimization methodology for thermal building simulation. The methodology will be based on the concept of consistent approximations [Pol97].

12 Conclusion

In system optimization, it is not possible to apply a general optimization algorithm that works efficiently on all problems. The efficiency of the optimization is strongly affected by what algorithm is used. The best algorithm to use depends on the properties of the objective function – such as the number of free parameters, the continuity of the objective function and its derivatives, and on the existence of local minima. Thus a variety of optimization algorithms is needed.

For optimizing the black-box functions that GenOpt is aimed at, a generalization of the structure of the optimization process can be made. First of all, the fact that analytical properties of the objective function are unavailable makes it possible to separate optimization and function evaluation. Therefore, a general interface is possible that allows coupling any stand-alone program that communicates via text files. With this approach, users are not restricted to using a special program for evaluating the objective function. Rather, they can use the simulation program they are already using for their system design and development. Hence, the system can be optimized with little additional effort.

This open environment not only allows you to couple your simulation program and implement special purpose algorithms, but it also allows sharing algorithms among users. This makes it possible to extend the algorithm library and thus improve GenOpt's usefulness.

To date, the optimization environment has been developed, different optimization algorithms have been implemented and various test cases have been run. A key finding is that a lot of function evaluations are being made in the neighborhood of the minimum. This indicates that substantial computation time can be saved with improvements in this stage of the optimization process. Some techniques that speed up the optimization in this region should be studied. One possibility is to use hybrid methods, which switch from one optimization scheme to another if the first reaches the slow improvement stage. Another approach to reach the minimum faster could be to use a multidimensional quadratic function approximation – based on points already evaluated – in the neighborhood of the minimum. Switching to the other optimization scheme could be done automatically or by user intervention based on graphical display of the rate of convergence.

13 Acknowledgment

The development of GenOpt was sponsored by grants from the Swiss Academy of Engineering Sciences (SATW), the Swiss National Energy Fund (NEFF) and the Swiss National Science Foundation (SNF) and is supported by the US Department of Energy (DOE). I would like to thank these institutions for their generous support.

14 Notice

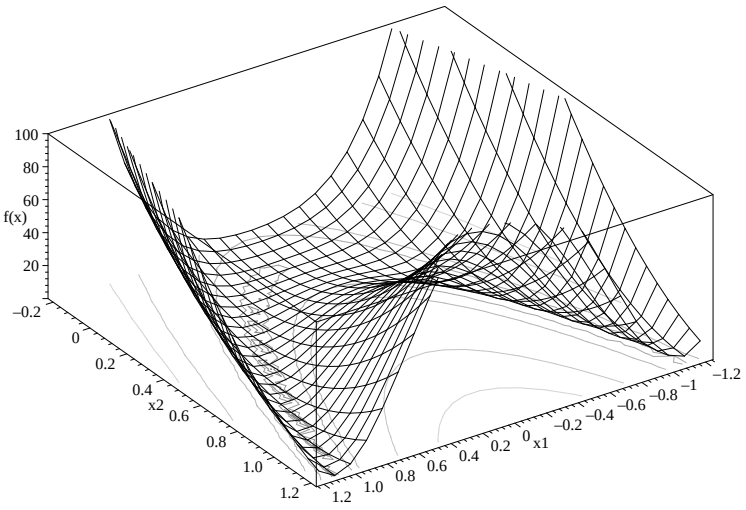


Fig. A.1: Rosenbrock function

A Benchmark Tests

This section lists the settings used in the benchmark tests.

The settings in `OptimizationsSettings` and `Algorithm` were the same for all runs expect for `Accuracy`, which is listed in the result chart on page 31.

The common settings were:

```
OptimizationSettings{
    MaxIte          = 1500;
    WriteStepNumber = false;
}

Algorithm{
   Main = NelderMeadONeill;
   Accuracy =  see page 31;
   StepSizeFactor = 0.001;
   BlockRestartCheck = 5;
   ModifyStoppingCriterion  =  see page 31;
}
```

The benchmark functions that were used and the `Parameter` settings in the `Vary` section are shown below.

A.1 Rosenbrock

The Rosenbrock function which is shown in Fig A.1 is defined as

$$f(x) \triangleq 100\left(x^2 - (x^1)^2\right)^2 + (1 - x^1)^2 \tag{A.1}$$

where $x \in \mathbb{R}^2$. It attains a minimum at $x^* = (1, 1)$, with $f(x^*) = 0$.

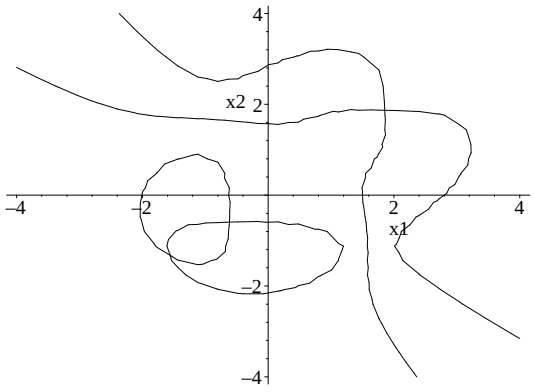


Fig. A.2: Contour plot of $\frac{\partial df(x)}{\partial x^1} = 0$ and $\frac{\partial df(x)}{\partial x^2} = 0$, where $f(x)$ is as in (A.2)

The section `Vary` of the optimization command file was set to

```
Vary{
   Parameter{
      Name =   x1; Min = SMALL;
      Ini  = -1.2; Max = BIG;
      Step =    1;
      }
   Parameter{
      Name =   x2; Min = SMALL;
      Ini  =    1; Max = BIG;
      Step =    1;
      }
}
```

A.2 Function 2D1

This function, which is composed of three other functions, has only one minimum point and therefore no other local minima. It has two regions where the gradient is very small (see Fig. A.2).

The function is defined by

$$f(x) \triangleq \sum_{i=1}^{3} f^i(x), \tag{A.2}$$

with

$$f^1(x) \triangleq \langle b, x\rangle + \frac{1}{2}\langle x, Q\,x\rangle, \quad b \triangleq \begin{pmatrix}1\\2\end{pmatrix}, \quad Q \triangleq \begin{pmatrix}10 & 6\\6 & 8\end{pmatrix}, \tag{A.3}$$

$$f^2(x) \triangleq 100\,\arctan\big((2-x^1)^2 + (2-x^2)^2\big), \tag{A.4}$$

$$f^3(x) \triangleq 50\,\arctan\big((0.5+x^1)^2 + (0.5+x^2)^2\big), \tag{A.5}$$

where $x \in \mathbb{R}^2$. The function has a minimum at $x^* = (1.855340, 1.868832)$ with $f(x^*) = -12.681271$.

The section `Vary` of the optimization command file is

```
Vary{
   Parameter{
      Name = x0; Min = SMALL;
      Ini  = -3; Max = BIG;
      Step = 0.1;
   }
   Parameter{
      Name =  x1; Min = SMALL;
      Ini  =  -3; Max = BIG;
      Step = 0.1;
   }
}
```

A.3 Function Quad

The function "Quad" is a 10-dimensional quadratic function of the form

$$f(x) \triangleq \langle b,\, x\rangle + \frac{1}{2}\langle x,\, M\, x\rangle, \tag{A.6}$$

with $b, x \in \mathbb{R}^{10}$ and $M \in \mathbb{R}^{10\times 10}$. The vector b is defined as

$$b \triangleq (10,\, 10,\, \ldots,\, 10). \tag{A.7}$$

This function is used in the benchmark test with two different matrices, M. Both matrices are positive definite.

In one test case, M is the identity matrix, I, and in the other test case M is a matrix – called Q (defined below) – with a large range of eigenvalues. The latter case is particularly interesting for testing Newton-based algorithms.

The matrix Q is defined as

579.7818	−227.6855	49.2126	−60.3045	−152.4101	−207.2424	8.0917	33.6562	204.1312	−3.7129
−227.6855	236.2505	−16.7689	−40.3592	179.8471	80.0880	−64.8326	15.2262	−92.2572	40.7367
49.2126	−16.7689	84.1037	−71.0547	20.4327	5.1911	−58.7067	−36.1088	−62.7296	7.3676
−60.3045	−40.3592	−71.0547	170.3128	−140.0148	8.9436	26.7365	125.8567	62.3607	−21.9523
−152.4101	179.8471	20.4327	−140.0148	301.2494	45.5550	−31.3547	−95.8025	−164.7464	40.1319
−207.2424	80.0880	5.1911	8.9436	45.5550	178.5194	22.9953	−39.6349	−88.1826	−29.1089
8.0917	−64.8326	−58.7067	26.7365	−31.3547	22.9953	124.4208	−43.5141	75.5865	−32.2344
33.6562	15.2262	−36.1088	125.8567	−95.8025	−39.6349	−43.5141	261.7592	86.8136	22.9873
204.1312	−92.2572	−62.7296	62.3607	−164.7464	−88.1826	75.5865	86.8136	265.3525	−1.6500
−3.7129	40.7367	7.3676	−21.9523	40.1319	−29.1089	−32.2344	22.9873	−1.6500	49.2499

and has eigenvalues in the range of 1 to 1000.

The functions achieve minimum points, x^*, at

Matrix M:	I	Q
x^{*^0}	−10	−2235.1810
x^{*^1}	−10	−1102.4510
x^{*^2}	−10	790.6100
x^{*^3}	−10	−605.2480
x^{*^4}	−10	−28.8760
x^{*^5}	−10	228.7640
x^{*^6}	−10	−271.8830
x^{*^7}	−10	−3312.3890
x^{*^8}	−10	−2846.7870
x^{*^9}	−10	−718.1490
$f(x^*)$	−500	0

Both test functions have been optimized with the same parameter settings. The settings for the parameters `x0` to `x9` are all the same:

```
Vary{
   Parameter{
      Name = x0; Min = SMALL;
      Ini  =  0; Max = BIG;
      Step =  1;
   }
}
```

Bibliography

[AD00] Charles Audet and J. E. Dennis, Jr. Analysis of generalized pattern search. Technical Report TR 00-07, Rice University, Houston, Department of Computational and Applied Mathematics, 2000.

[Avr76] Mordecai Avriel. *Nonlinear programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

[BP66] M. Bell and M. C. Pike. Remark on algorithm 178. *Comm. ACM*, 9:685–686, Sep. 1966.

[DS83] John E. Dennis, Jr. and Robert B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1983.

[DV68] R. De Vogelaere. Remark on algorithm 178. *Comm. ACM*, 11:498, Jul. 1968.

[FR64] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *Comput. J.*, 7:149–154, 1964.

[GMW81] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical optimization*. Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London, 1981.

[HJ61] R. Hooke and T. A. Jeeves. 'Direct search' solution of numerical and statistical problems. *J. Assoc. Comp. Mach.*, 8(2):212–229, Apr. 1961.

[Kar89] V. G. Karmanov. *Mathematical programming*. "Mir", Moscow, 1989.

[LRWW98] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9(1):112–147, 1998.

[LT99] Robert Michael Lewis and Virginia Torczon. Pattern search algorithms for bound constrained minimization. *SIAM Journal on Optimization*, 9(4):1082–1099, 1999.

[LTT00] Robert Michael Lewis, Virginia Torczon, and Michael W. Trosset. Direct search methods: Then and now. Technical Report TR-2000-26, NASA, ICASE, Langley Research Center, Hampton, VA, May 2000.

[McK98] K. I. M. McKinnon. Convergence of the Nelder-Mead simplex method to a nonstationary point. *SIAM Journal on Optimization*, 9(1):148–158, 1998.

[NM65] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, Jan. 1965.

[O'N71] R. O'Neill. Algorithm AS 47-function minimization using a simplex procedure. *Appl. Stat. 20*, 20:338–345, 1971.

[Pol97] Elijah Polak. *Optimization, Algorithms and Consistent Approximations*, volume 124 of *Applied Mathematical Sciences*. Springer Verlag, 1997.

[Pow64] M. J. D. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal*, 7(2):155–162, Jul. 1964.

[Pow65] M. J. D. Powell. A method for minimizing a sum of squares of nonlinear functions without calculating derivatives. *The Computer Journal*, 7(4):303–307, 1965.

[PSU88] Anthony L. Peressini, Francis E. Sullivan, and J. J. Uhl, Jr. *The mathematics of nonlinear programming*. Springer-Verlag, New York, 1988.

[Sch94] Hans Paul Schwefel. *Evolution and optimum seeking*. John Wiley & Sons, Inc., 1994.

[Smi69] Lyle B. Smith. Remark on algorithm 178. *Comm. ACM*, 12:638, Nov. 1969.

[Tor97] Virginia Torczon. On the convergence of pattern search algorithms. *SIAM Journal on Optimization*, 7(1):1–25, 1997.

[Wal75] G. R. Walsh. *Methods of optimization*. Wiley-Interscience [John Wiley & Sons], London, 1975.

[Wil64] D. J. Wilde. *Optimum seeking methods*. Prentice-Hall, USA, 1964.

[Wri96] M. H. Wright. Direct search methods: once scorned, now respectable. In D. F. Griffiths and G. A. Watson, editors, *Numerical Analysis 1995*, pages 191–208. Addison Wesley Longman (Harlow), 1996.